Applications Development for the Computational Grid

David Abramson
ARC Professorial Fellow
Professor of Computer Science
Monash University
AUSTRALIA

Summary

e-Science, enabled by the emerging Grid computing paradigm, tightly couples scientists, their instruments (e.g., telescopes, synchrotrons, and networks of sensors), massive data storage devices and powerful computational devices. This new discipline allows scientists to interact efficiently and effectively with each other, their instruments and their data, even across geographic separations, thereby ameliorating the tyranny of distance that often hinders research. Data can be captured, shared, interpreted and manipulated more efficiently and more reliably and on a far greater scale than previously possible. Data can be presented for interpretation in new ways using scientific visualization techniques and advanced data mining algorithms. These new technologies enable new insights to be derived and exploited. The data may also drive simulation models that support prediction and "what-if" analyses. The models and their results may be archived for later use and analysis, and shared securely and reliably with scientific collaborators across the globe. The resulting network of people and devices is empowered to interact more productively and to undertake experiments and analyses that are otherwise impossible.

In spite of tremendous advances in middleware and Internet software standards, creating Grid applications that harness geographically disparate resources is still difficult and error-prone. Programmers are presented with a range of middleware services, a raft of legacy software tools that do not address the distributed nature of the Grid, and many other incompatible development tools that often deal with only part of the Grid programming problem. So, a scientist might start with an idea for an innovative experiment but quickly become distracted by technical details that have little to do with the task at hand. Moreover, the highly distributed, heterogeneous and unreliable nature of the Grid makes software development extremely difficult. If we are to capitalize on the enormous potential offered by Grid computing, we must find more efficient and effective ways of developing Grid based applications.

A critical ingredient for success in e-Science is appropriate Grid-enabled software which, to date, has lagged behind the high-performance computers, data servers, instruments and networking infrastructure.

All software follows a lifecycle, from development through execution, and back again. Grid software is no exception, although there are sufficient differences in the details of the various phases in the lifecycle to make traditional tools and techniques inappropriate. For example, traditional software development tools rarely support the creation of virtual applications in which the components are distributed across multiple machines. In the Grid, these types of virtual applications are the norm. Likewise, traditional methods of debugging software do not scale to the size and heterogeneity of infrastructure found the Grid. Here, we identify four distinct phases of importance, development, deployment, testing and debugging and execution.

Typically, e-Science applications use services that are exposed by both the platform infrastructure and middleware. In our experience, whilst powerful, these services are typically too low level for many e-Science applications. As a result, there is a significant 'semantic gap' between them, because the application needs are not matched by the underlying middleware services. Moreover, they do not support the software lifecycle thereby making software development difficult and error-prone.

To solve these problems, we propose a new hierarchy in which existing middleware is renamed lower-middleware, and an upper middleware layer is inserted. This upper middleware layer is designed to narrow the semantic gap between existing middleware and applications. Importantly, it hosts a range of interoperating tools that will form the e-Scientists workbench, thus supporting the major phases of the software development lifecycle as well as the applications themselves.

This Doctorate of Science thesis tracks the development, since the mid 1990's, in designing solutions that support the software lifecycle of Grid software. Under my supervision, my research students and associates have designed new approaches, and built exemplars that demonstrate the effectiveness of the ideas. We have applied these tools to real world e-Science problems.

The work is divided into 14 chapters that focus on issues in the development, deployment, debugging and execution of e-Science applications. The thesis tells a story, moving from earlier, less sophisticated, ideas and tools, through to more advanced and effective approaches. Each chapter deals with a particular approach, and highlights the software tools that embody the ideas.

Chapter 1 provides an overall framework in which the techniques and software tools support the Grid software life cycle. A slightly modified framework is presented in pictorial form as a template for the remaining chapters. Chapter 2 presents work on parametric computing, and its implementation in a tool called Nimrod. Chapter 3 extends the resource base for parametric computing to encompass globally distributed machines. Chapter 4 discusses the use of non-linear optimization algorithms to guide parametric search. Chapter 5 shows how to apply a technique called Fractional Factorial Design to the parametric computing problem. Chapter 6 develops a richer framework and shows how parametric computing can be embedded in scientific workflows. Chapter 7 proposes a novel technique that allows users to specify their parametric experiments via spreadsheets. Chapter 8 presents additional papers that single out particular applications not already shown in earlier papers. Chapter 9 discusses issues in building real production Grid testbeds. Chapter 10 discusses relative debugging, and its application to Grid software engineering. Chapter 11 discusses techniques for replicating data on the Grid and develops a new model called Active Data. Chapter 12 discusses general tools for performing debugging across the distributed Grid infrastructure. Chapter 13 presents techniques for solving firewall and security issues in real Grids. Chapter 14 presents techniques for deploying software applications on the Grid. Chapter 15 lists work that has not been included because it is too recent.

The collected chapters both propose new techniques and strategies, but also illustrate these with real software solutions. As a result, new tools, such as the Nimrod family of development tools and the Guard debugging tools, have been built and applied to real applications. Thus, this work has demonstrates its applicability not only through a set of well-cited papers, but also by user adoption.

Extent to which the work contributes to the advancement of knowledge

As discussed, this thesis starts by building a new framework for Grid software, and then proceeds to outline a range of techniques and tools that allow programmers to build, deploy, test and execute real applications. The introductory chapter proposes a split middleware stack for the first time. Each of the approaches discussed, for example, parametric computing, relative debugging, Grid debugging, active spreadsheets, testbed management, active data and application deployment have made original contributions to the state of the art. The work forms a cohesive body over an extended period, but stitches the ideas together into a simple and visionary solution. Some of the ideas have found their way into commercial application – most notably, the Nimrod tools exist as a commercial product called EnFuzion (marketed by Axceleon Inc) and the Guard debugger has been licensed to Cray Research.

Statement of contribution

Computer Science, and e-Science in general, is a collaborative venture, and thus, almost all of the papers selected here have multiple authors. As the lead researchers, most of the work presented here is built on my ideas, implemented by various research associates employed on my research grants. Some ideas have been jointly developed with PhD students. However, even in these latter cases, I made significant and substantial contributions to the work. The coherence and consistency of the themes across and between students demonstrates a long-term view of the overall research theme, namely, the development of new techniques that allow users to leverage the global Grid. In each chapter I identify individual contributions in more detail.

Dedication

This work is dedicated to my family. Without their patience, love and support, I would not have been able to invest the substantial time required to complete it.

I particularly want to acknowledge my wife, Heather, who has been by fellow traveler on this long journey. Without her love and support, my work would have never seen the light of day.

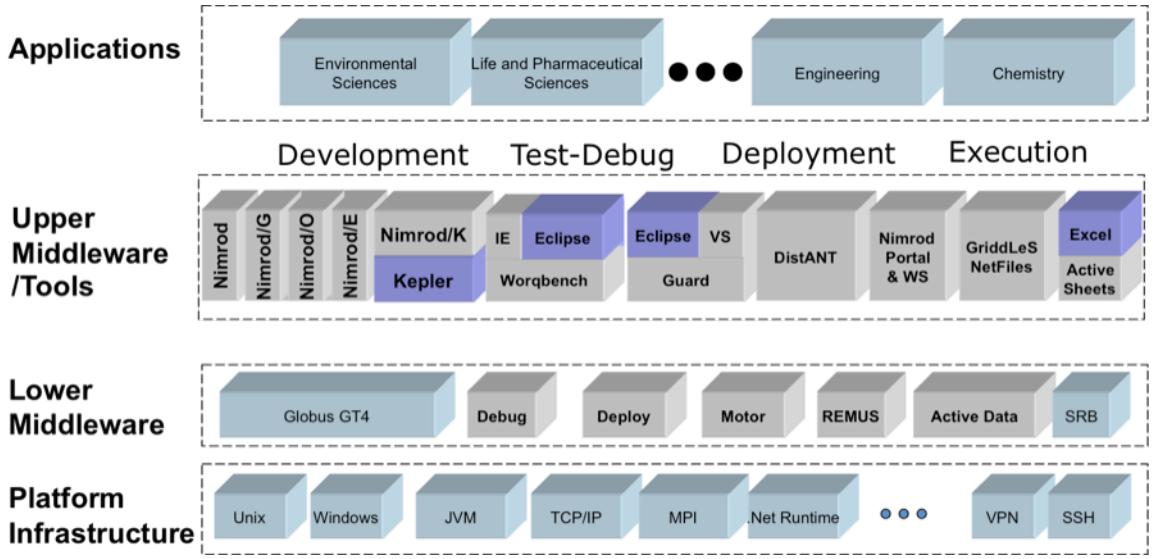
My three beautiful daughters, with their unique strengths and personalities, are only just beginning their professional lives. I hope this volume inspires them to pursue their dreams and contribute in every possible way. I greatly admire their achievements to date, and look forward to watching them continue to grow.

Chapter 1: Grid Middleware

In this chapter I discuss an overarching framework for the research presented in this collection. This sole authored paper was an invited keynote address the Eighth Asia Pacific Web Conference in Harbin, China, 2006.

(1) Abramson, D. "Applications Development for the Computational Grid", The Eighth Asia Pacific Web Conference, Harbin, China, Invited Key Note Address, 16th – 18th Jan, 2006. Lecture Notes in Computer Science, Volume 3841 / 2006, pp. 1 – 12, ISSN: 0302-9743: 1-1 … 1-12

Since 2006 I have refined the architecture, and the following picture provides a more accurate summary of the components of the framework. This framework will be used throughout the thesis to illustrate the contribution of the various components.



Applications Development for the Computational Grid

David Abramson

Faculty of Information Technology
Monash University,
Clayton, Vic, Australia, 3800
davida@csse.monash.edu.au
http://www.csse.monash.edu.au/~davida

Abstract. The Computational Grid has promised a great deal in support of innovative applications, particularly in science and engineering. However, developing applications for this highly distributed, and often faulty, infrastructure can be demanding. Often it can take as long to set up a computational experiment as it does to execute it. Clearly we need to be more efficient if the Grid is to deliver useful results to applications scientists and engineers. In this paper I will present a raft of upper middleware services and tools aimed at solving the software engineering challenges in building real applications.

1 Introduction

e-Science, enabled by the emerging Grid computing paradigm [28], tightly couples scientists, their instruments (e.g., telescopes, synchrotrons, and networks of sensors), massive data storage devices and powerful computational devices. This new discipline allows scientists to interact efficiently and effectively with each other, their instruments and their data, even across geographic separations, thereby ameliorating the tyranny of distance that often hinders research. Data can be captured, shared, interpreted and manipulated more efficiently and more reliably and on a far greater scale than previously possible. Data can be presented for interpretation in new ways using scientific visualization techniques and advanced data mining algorithms. These new technologies enable new insights to be derived and exploited. The data may also drive simulation models that support prediction and "what-if" analyses. The models and their results may be archived for later use and analysis, and shared securely and reliably with scientific collaborators across the globe. The resulting network of people and devices is empowered to interact more productively and to undertake experiments and analyses that are otherwise impossible.

In spite of tremendous advances in middleware and internet software standards, creating Grid applications that harness geographically disparate resources is still ifficult and error-prone. Programmers are presented with a range of middleware services, a raft of legacy software tools that do not address the distributed nature of the Grid, and many other incompatible development tools that often deal with only part of the Grid programming problem. So, a scientist might start with an idea for an innovative experiment but quickly become distracted by technical details that have little

to do with the task at hand. Moreover, the highly distributed, heterogeneous and unreliable nature of the Grid makes software development extremely difficult. If we are to capitalize on the enormous potential offered by Grid computing, we must find more efficient and effective ways of developing Grid based applications.

2. Software engineering for the Grid

A critical ingredient for success in e-Science is appropriate Grid-enabled software which, to date, has lagged behind the high-performance computers, data servers, instruments and networking infrastructure.

All software follows a lifecycle, from development through execution, and back again, (Figure 1). Grid software is no exception, although there are sufficient differences in the details of the various phases in the lifecycle to make traditional tools and techniques inappropriate. For example, traditional software development tools rarely support the creation of virtual applications in which the components are distributed across multiple machines. In the Grid, these types of virtual applications are the norm. Likewise, traditional methods of debugging software do not scale to the size and heterogeneity of infrastructure found the Grid. Here, we identify four distinct phases of importance, development, deployment, testing and debugging and execution.

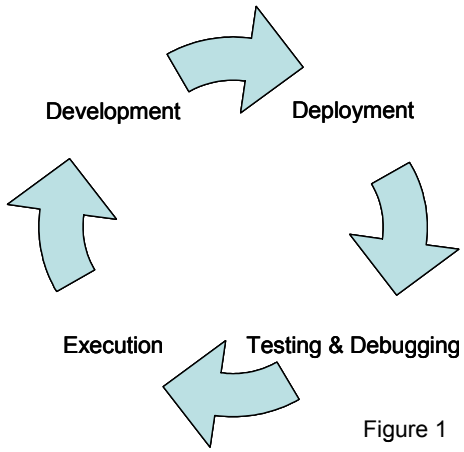


Figure 1

2.1 Development

Initially, software is developed using the most appropriate tools and programming languages for the task at hand. The process involves the specification, coding and compilation of the software. In the Grid, there is a very strong focus on building "virtual applications", or workflows, that consist of a number of interoperating components distributed across multiple resources. Grid workflows are powerful because

they support the integration of computations, data, scientific instruments and visualization-software while leveraging multiple Grid resources. Grid workflows have been specified for many different scientific domains including physics [31] gravitational wave physics [25], geophysics [40] , astronomy [15] and bioinformatics [36]. Accordingly, there have been many projects to date that support Grid workflows, to name a few, *Triana* [47], *Taverna* [37][42], *Kepler* [35], *GrADS* [17] and *P-Grade* [33].

A specialized form of workflow allows the creation of "parameter sweeps", where a computational model is run repeatedly with different input parameters. Using this approach it is possible to explore different design options, and perform more robust science than previously possible. A number of systems support parameter sweep workflows, including APST [21], the NASA IPG (Information Power Grid) parameter process specification tool [50] and our own Nimrod/G [1][4][5][7].

Apart from these specific environments, programmers can adopt any one of a number of techniques for building distributed applications. These might build on standard languages like Java, and may use special message passing libraries like MPICH/G. Further, tools like Ninf-G [46] and NetSolve [22] provide powerful remote procedure call systems that can invoke arbitrary procedures as well as specific services such as linear algebra. Finally, Web Services provide a generic and standards based mechanism for building large distributed applications, and these underpin the newly developed Web Services Resource Framework (WSRF) [24].

2.2 Deployment

Traditionally, deployment is often combined with development, and may involve little more than copying the executable program to some partition in the file system, and perhaps registering the software with the operating system. The difficulty in deployment is often underestimated because modern installation techniques, such as those used by Microsoft and Apple, appear to simplify the process enormously. However, in a Grid environment, deployment is much more complex because of the need to install the code on a range of heterogeneous machines, geographically distributed and in different administrative domains. In the Grid deploying an application means building and installing the software on a range of different platforms, taking account of issues such as different instruction sets, operating systems, file system structures and software libraries. To date, this phase is often performed manually, which is both error-prone and does not scale to large Grids. For example, in order to deploy an application across 500 computers, a user would typically need to log into each of the 500 sequentially, compiling, linking and installing the software. Our own experiences at deploying a quantum chemistry package over a handful of resources have identified this as a serious bottleneck [43][44].

Surprisingly, there has been little work on the deployment problem, and none of the current middleware projects addresses deployment across heterogeneous resources. Some researchers have suggested solving the problem by taking a system-centric view, as is done in systems that configure a homogeneous compute cluster [18][14][29][30][13]. In this model, an application image is produced for the entire

Grid, and then copied from a central resource. However, this central approach is in conflict with the philosophy of the Grid, which favours a decentralized approach. Moreover, it does not handle significant heterogeneity because each resource could, in the worst case, require a tailored version of the software. One of the few systems that views deployment in a decentralized way is GridAnt [48].

2.3 Testing and Debugging

Testing and debugging software is already challenging on a local workstation. In the Grid this phase is particularly difficult, because the software must be tested and debugged on a range of different platforms, distributed geographically and in different administrative domains. Traditional testing and debugging tools are unable to provide the support required. At present, the only feasible way of debugging a piece of Grid software is for the programmer to log into the remote system and run a conventional debugger such as *gdb* [41]. This technique does not scale to large Grids, and is not practical for workflows that are distributed across multiple resources. Debugging is a serious challenge in the Grid, partly because an application must often execute on a range of different platforms. In this environment, it is not uncommon for a program to work correctly on one machine, but fail in subtle ways when the software is ported or moved to another platform. Traditional debugging techniques usually force the programmer to debug the ported application from scratch, machine the task complex and time consuming.

2.4 Execution

This phase typically means scheduling and coordinating execution, using a variety of resources concurrently. Of the phases discussed to date, execution has attracted the most attention. There are many different ways of starting, scheduling and controlling the execution of Grid software, ranging from direct interface to the middleware through to sophisticated scheduling and orchestration systems.

Web Services and Globus [27] provide rudimentary mechanisms for starting jobs on a given remote, and these services can build more complex multi-resource schedules. For example, Kepler contains methods for scheduling and controlling the execution of a workflow and uses the Globus GRAM interface to execute the various workflow actors [12]. Other middleware, such as Condor-G [26][34] and APST [21], use sophisticated scheduling algorithms to enforce quality of service metrics. For example, APST can minimize the total execution time of a workflow. Systems such as Cactus [11] provide techniques for migrating computations across Grid resources, thus the computation adapts to the variability in the resource availability.

3. Grid Middleware

Figure 2 shows a traditional software hierarchy. Here, e-Science applications use services that are exposed by both the platform infrastructure and middleware such as Globus and Unicore [39]. In our experience, whilst powerful, these services are typically too low level for many e-Science applications. As a result, there is a significant 'semantic gap' between them, because the application needs are not matched by the underlying middleware services. Moreover, they do not support the software lifecycle thereby making software development difficult and error-prone.

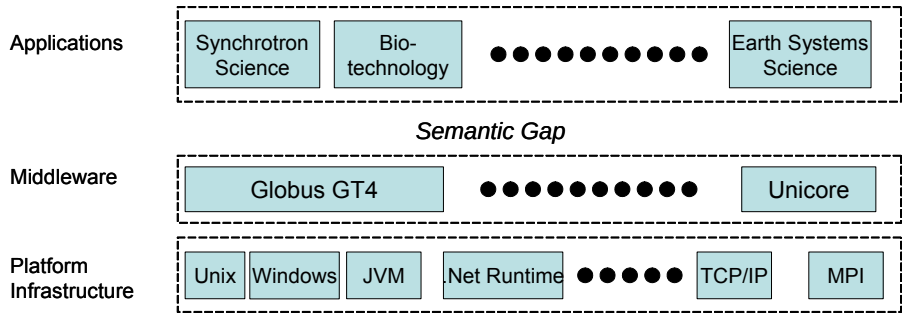


Figure 2 – Ttraditional software hierarchy

To solve these problems, we propose a new hierarchy as shown in Figure 3. The existing middleware is renamed lower-middleware, and an upper middleware layer is inserted. This upper middleware layer is designed to narrow the semantic gap between existing middleware and applications. Importantly, it hosts a range of interoperating tools that will form the e-Scientists workbench, thus supporting the major phases of the software development lifecycle as well as the applications themselves.

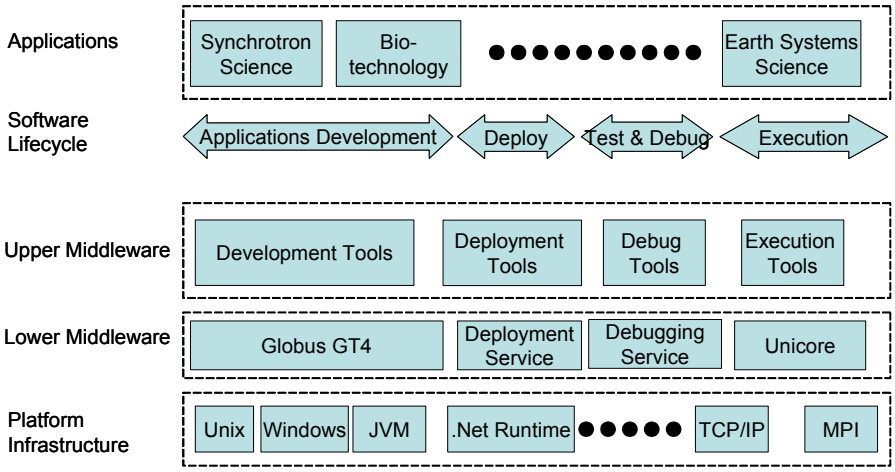


Figure 3 – New software hierarchy

4. Upper Middleware and Tools

Our research group has built a number of software tools that address some of the challenges sited in Section 2, as shown in Figure 4. In particular, Nimrod and GriddleS target software development; Guard focuses on debugging; Grid Work Bench and DistAnt target deployment and Nimrod, GriddLeS, Active Sheets, REMUS and the Nimrod Portal all focus on execution. In this section we provide a very brief overview of these tools.

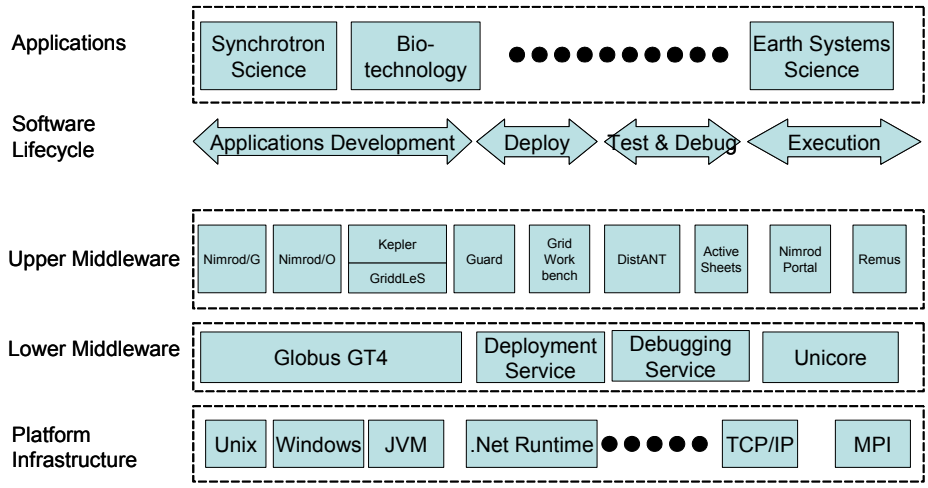


Figure 4 – Monash Grid Tools

4.1 Development

Nimrod/G and GriddLeS [6][8], address some of the challenges in creating of Grid software. Nimrod/G manages the execution of studies with varying parameters across distributed computers. It takes responsibility for the overall management of an experiment as well as the low-level issues of distributing files to remote systems, performing the remote computations, and gathering the results. When users describe an experiment to Nimrod/G, a declarative plan file is developed that describes the parameters, their default values, and the commands needed to perform the work. Apart from this high-level description, users are freed from much of the complexity of the Grid. As a result, Nimrod/G has been very popular among application scientists. Nimrod/O is a variant of Nimrod/G that performs a guided search of the design space rather than exploring all combinations. Nimrod/O allows users to phrase questions such as: “What set of design parameters will minimize (or maximize) the output of my model?” If the model computes metrics such as cost and lifetime, it is then possible to perform automatic optimal design. A commercial version of Nimrod, called EnFuzion, has been produced [16].

GriddLeS, on the other hand, provides a very flexible input-output model that makes it possible to build workflows from legacy applications (written in Fortran, C,

etc) thereby leveraging the enormous amount of scientific software that already exists. GriddLeS allows existing programs to transparently access local and remote files, as well as data that is replicated across multiple servers using Grid middleware such as the Storage Resource Broker [38] and the Globus Replica Location Service [23]. It also allows workflows to pipe data from one application to another without any changes to the underlying data access model. In order to support scientific workflows, we have coupled GriddLeS with the Kepler workflow system. Kepler is an active open source cross-project, cross-institution collaboration to build and evolve a scientific workflow system on top of the Ptolemy II system. Kepler allows scientists from multiple domains to design and execute scientific workflows. It includes two dataflow-based computation models, Process Networks (PN) and Synchronous Data Flow (SDF), and these can be used to define the "orchestration semantics" of a workflow. Simply by changing these models, one can change the scheduling and overall execution semantics of a workflow. By combining Kepler and GriddLeS, a user has significant flexibility in choosing the way data is transferred between the individual components, and this can be done without any changes to the application source.

4.2 Deployment

We are currently developing a few different tools to solve the deployment problem, specifically DistAnt and GWB. DistAnt provides an automated application deployment system with a user-oriented approach [30]. It is targeted at users with reasonable knowledge of the application they are deploying, but strictly limited grid computing knowledge, resource information and importantly, resource authority. DistAnt provides a simple, scalable and secure deployment service and supports a simple procedural deployment description.

DistAnt supports application deployment over heterogeneous grids by virtualizing certain grid resource attributes to provide a common application deployment gateway, deployment description, file system structure and resource description. To manage remaining resource heterogeneity DistAnt supports sub-grids, to divide an unmanageable heterogeneous grid into manageable sets of like resources, categorized by resource attributes that can be queried. Sub-grids provide a framework to execute environment specific remote build routines, compile an application over a set of resource platforms and redistribute binaries to the entire grid. DistAnt also supports definition and deployment of application dependencies. DistAnt enables deployment of a complex native application over an uncharacterized heterogeneous grid, assuming nothing about grid resources.

Furthermore, integration of DistAnt into Nimrod/G, provides an overall environment enabling grid scale application development, deployment and execution.

In addition to DistAnt, we are building a rich interactive development environment (IDE), called Grid Work Bench (GWB). GWB is based on the public domain platform Eclipse [32], and supports the creation, management, distribution and debugging of Grid applications. GWB provides specific functionality to help programmers manage the complexity and heterogeneity of the Grid.

4.3 Testing and Debugging

The Guard debugger targets the process of testing and debugging in the Grid [2][3]. Specifically, it solves some of the problems discussed in Section 2.3 concerning programs that fail when they are ported from one Grid resource to another. We use a new methodology called relative debugging, which allows users to compare data between two programs being executed. Relative Debugging is effectively a hybrid test-and-debug methodology. While traditional debuggers force the programmer to understand the expected state and internal operation of a program, relative debugging makes it possible to trace errors by comparing the contents of data structures between programs at run time. In this way, programmers are less concerned with the actual state of the program. They are more concerned with finding when, and where, differences occur between the old and new code. The methodology requires users to begin by observing that two programs generate different results. They then move back iteratively through the data flow of the codes, to determine the point at which different answers appear. Guard supports the execution of both sequential and parallel programs on a range of platforms. It also exists for a number of different development environments. Because Guard uses a client-server architecture, it is possible to run a debug client on one Grid resource and have it debug an application running on another one, removing the need for users to log into the target system. Further, Guard uses a platform neutral data representation called AIF [49] which means the client and debug servers can run on different types of architecture.

We are concurrently developing a WSRF compliant debug service that will allow high level tools like the GRB to debug applications across multiple Grid resources. This debug service will interoperate with the Globus GRAM interface, thus jobs launched by the GRAM can be debugged in a secure and efficient way using the additional interface.

4.4 Execution

Nimrod provides significant support during the execution of parameter sweeps, including a sophisticated scheduler that enforces real time deadlines. This economy allows users to specify soft real time deadlines that are enforced by trading units in a computational economy [19][20]. Using this approach the system can provide a quality of service that is proportional to the amount of currency a user wishes to expend on an experiment – in short, the more a user pays, the more likely they are to meet their deadline at the expense of another user. The Nimrod scheduler supports two types of inter-task constraints, namely parallel and sequential dependencies. Parallel tasks are executed concurrently providing there are sufficient computational resources. Typically, these tasks pertain to different parameter values in a parameter sweep and can be executed in parallel. However, it is possible to specify special sequential parameters (called seqameters, as opposed to parameters) that force the order of the execution to be sequential. This means that one task may be dependent on the output from another, and its execution can be stalled until the data is available.

The Nimrod Portal and Active Sheets address the execution phase of the life cycle. The Nimrod Portal allows users to create Nimrod experiments from a web interface.

It supports the creation of the plan files discussed above using a graphical user interface, the management of the test bed (and associated Globus issues such as certificate management), and control of the experiment as it executes. Active Sheets [10] allows users to set up and execute an experiment from a familiar spreadsheet interface. Individual cells can invoke Nimrod/G to perform one simulation run; multiple data independent cells can be used to specify an entire "what if" experiment. Because the system is embedded in Microsoft Excel, all normal data manipulation and charting tools are available for post analysis (a feature that is popular with users).

REMUS is an execution environment that helps users build complex Grid applications across firewalls and different administrative domains [45]. REMUS provides mechanisms that reroute traffic through approved channels without compromising the security of any site. It effectively handles heterogeneity in security mechanisms, allowing applications to communication when there is no common security framework.

5. Conclusion

In this paper we have provided a very brief overview of the challenges in building software for the Grid. We have focused on four phases of a software lifecycle, namely development, deployment, testing and debugging. We have shown that it is possible to split Grid middleware into two layers – one that addresses low level issues and a higher application focused layer. This latter layer can support software tools that make the software development task easier. We have discussed a number of tools developed by the author that simplify the software development task.

Acknowledgments

The author wishes to acknowledge a number of people who have contributed to the work described in this document, including Shahaan Ayyub, Rajkumar Buyya, Phillip Chan, Clement Chu, Colin Enticott, Jagan Kommineni, Donny Kurniawan, Slavisa Garic, Jon Giddy, Wojtek Goscinski, Tim Ho, Andrew Lewis, Tom Peachey, Jeff Tan and Greg Watson.

The projects are supported by a variety of funding agencies, including the Australian Research Council, the Australian Department of Communications, Arts and Information Technology (DCITA), the Australian Department of Education, Science and Technology (DEST), Microsoft, IBM and Hewlett Packard.

References

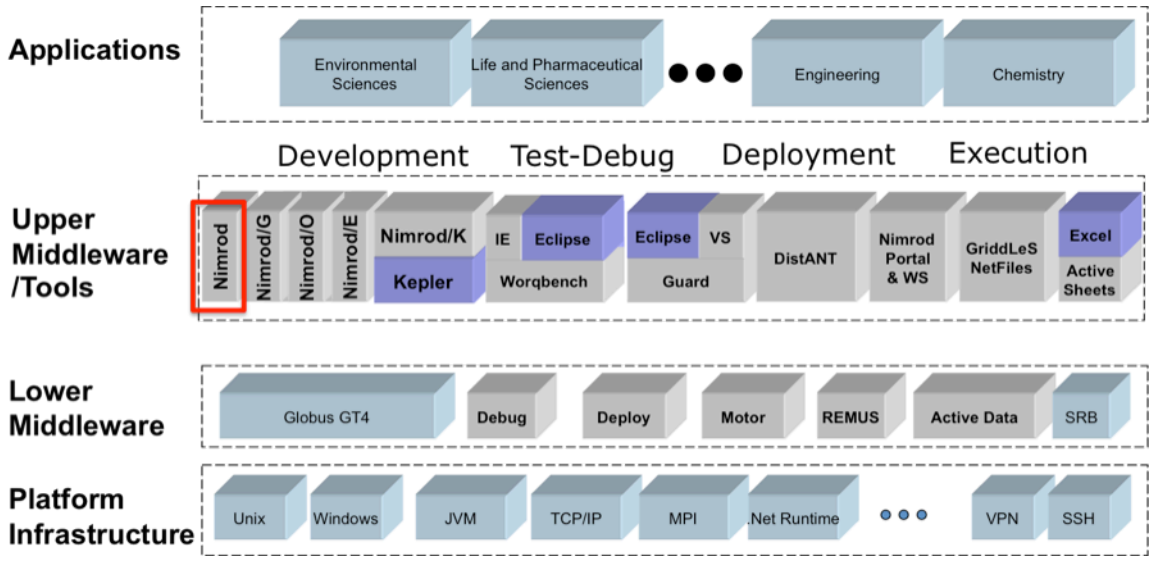
1 Abramson D, Lewis A, Peachey T, Fletcher, C., "An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics", SuperComputing 2001, Denver, Nov 2001.

2 Abramson D., Foster, I., Michalakes, J. and Sosic R., "Relative Debugging: A new paradigm for debugging scientific applications", Communications of the Association for Computing Machinery (CACM), Vol. 39, No 11, pp 67 - 77, Nov 1996.
3 Abramson D., Foster, I., Michalakes, J. and Sosic R., "Relative Debugging and its Application to the Development of Large Numerical Models", Proceedings of IEEE Supercomputing 1995, San Diego, December 95. Paper on CD, no pages
4 Abramson D., Sosic R., Giddy J. and Hall B., "Nimrod: A Tool for Performing Parametrised Simulations using Distributed Workstations", The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.
5 Abramson, D, Lewis, A. and Peachy, T., "Nimrod/O: A Tool for Automatic Design Optimization", The 4th International Conference on Algorithms & Architectures for Parallel Processing (ICA3PP 2000), Hong Kong, 11 - 13 December 2000.
6 Abramson, D. and Kommineni, J., "A Flexible IO Scheme for Grid Workflows". IPDPS-04, Santa Fe, New Mexico, April 2004.
7 Abramson, D., Giddy, J. and Kotler, L. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?, International Parallel and Distributed Processing Symposium (IPDPS), pp 520- 528, Cancun, Mexico, May 2000.
8 Abramson, D., Kommineni, J. and Altinas, I. "Flexible IO services in the Kepler Grid Workflow Tool", to appear, IEEE Conference on e-Science and Grid Computing, Melbourne, Dec 2005.
9 Abramson, D., Roe, P., Kotler L and Mather, D., "ActiveSheets: Super-Computing with Spreadsheets". 2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference, April 22-26, 2001, pp 110 – 115, Seattle, Washington (USA).
10 Abramson, D., Roe, P., Kotler L and Mather, D., "ActiveSheets: Super-Computing with Spreadsheets". 2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference, April 22-26, 2001, pp 110 – 115, Seattle, Washington (USA).
11 Allen G. and Seidel, E. "Collaborative Science: Astrophysics Requirements and Experiences", in The Grid: Blueprint for a New Computing Infrastructure (2nd Edition), Ed: Ian foster and Carl Kesselmann, p. 201-213, (2004)
12 Altintas, I. Berkley, C. Jaeger, E. Jones, M. Ludäscher B. and Mock, S. "Kepler: Towards a Grid-Enabled System for Scientific Workflows," in the Workflow in Grid Systems Workshop in GGF10 - The 10th Global Grid Forum, Berlin, March 2004.
13 Anderson P. and Scobie. A. "LCFG: The Next Genera-tion",UKUUG Winter Conference, 2002.
14 Anderson, P. Goldsack, P. Paterson, J. "SmartFrog meets LCFG - Autonomous Reconfiguration with Central Policy Control", 2002 Large Installations Systems Admini-stration Conference, 2003
15 Annis, J., Zhao,Y. et al., "Applying Chimera Virtual Data Concepts to Cluster Finding in the Sloan Sky Survey," Technical Report GriPhyN-2002-05, 2002.
16 Axceleon Inc, http://www.axceleon.com
17 Berman et al, "The GrADS Project: Software Support for High-Level Grid Application Development", International Journal of High Performance Computing Applications, Winter 2001 (Volume 15, Number 4), pp. 327-344.
18 Bruno, G. Papadopoulos P. and Katz., M. "Npaci rocks: Tools and techniques for easily deploying manageable linux clusters". Cluster 2001, 2001
19 Buyya, R., Abramson, D. and Giddy, J. "Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid", HPC Asia 2000, May 14-17, 2000, pp 283 – 289, Beijing, China

20 Buyya, R., Abramson, D. and Venugopal, S. “The Grid Economy”, Special Issue on Grid Computing, Proceedings of the IEEE, Manish Parashar and Craig Lee (editors), Volume 93, Issue 3, 698-714pp, IEEE Press, New York, USA, March 2005.
21 Casanova H. and Berman, F. “Parameter Sweeps on The Grid With APST”, chapter 26. Wiley Publisher, Inc., 2002. F. Berman, G. Fox, and T. Hey, editors.
22 Casanova H. and Dongarra, J., “Netsolve: A Network Server for Solving Computational Science Problems,” Supercomputing Applications and High Performance Computing, vol. 11, no. 3,pp. 212–223, 1997.
23 Chervenak, A.L. Palavalli, N. Bharathi, S. Kesselman C. and Schwartzkopf, R. “Performance and Scalability of a Replica Location Service,” Proceedings of the International IEEE Symposium on High Performance Distributed Computing (HPDC-13), June 2004
24 Czajkowski, K. Foster, I. Frey, J. et al. “The WS-Resource Framework, Version 1.0”, 03/05/2004, http://www.globus.org/wsrf/
25 Deelman, E., Blackburn, K. et al., "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists," 11th Intl Symposium on High Performance Distributed Computing, 2002.
26 Douglas Thain, Todd Tannenbaum, and Miron Livny, "Condor and the Grid", in Fran Berman, Anthony J.G. Hey, Geoffrey Fox, editors, Grid Computing: Making The Global Infrastructure a Reality, John Wiley, 2003.
27 Foster I. and Kesselman C., Globus: A Metacomputing Infrastructure Toolkit, International Journal of Supercomputer Applications, 11(2): 115-128, 1997.
28 Foster, I and Kesselman, C. (editors), The Grid: Blueprint for a New Computing Infrastructure, ISBN: 1558609334, 2nd edition, Morgan Kaufmann, USA, November 18, 2003.
29 Goldsack, P. Guijarro, J. Lain, A. et al, “SmartFrog: Configuration and Automatic Ignition of Distributed Applications”, HP Labs, UK, 2003, http://www.hpl.hp.com/research/smartfrog/
30 Goscinski, W. Abramson, “D. Distributed Ant: A System to Support Application Deployment in the Grid”, Fifth IEEE/ACM International Workshop on Grid Computing, 2004.
31 GriPhyN, www.griphyn.org
32 http://www.eclipse.org
33 Kacsuk, P., Cunha, J.C., Dózsa, G., Lourenco, J., Antao, T., Fadgyas, T., “GRADE: A Graphical Development and Debugging Environment for Parallel Programs”, Parallel Computing Journal, Elsevier, Vol. 22, No. 13, Feb. 1997, pp. 1747-1770
34 Litzkow, M., Livny, M. and Mutka, M. Condor - A Hunter of Idle Workstations". Proceedings of the 8th International Conference on Distributed Computing Systems. San Jose, Calif., June, 1988.
35 Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M, Lee, E., Tao J. and Zhao, Y. “Scientific Workflow Management and the Kepler System”, Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows, 2005.
36 NPACI, "Telescience, https://gridport.npaci.edu/Telescience/."
37 Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M., Wipat, A. and Li., P. “Taverna: A tool for the composition and enactment of bioinformatics workflows”, Bioinformatics Journal 20(17) pp 3045-3054, 2004, doi:10.1093/bioinformatics/bth361.
38 Rajasekar, A. Wan, M. Moore, R. Schroeder, W. Kremenek, G. Jagatheesan, A. Cowart, C. Zhu, B. Chen S. and Olschanowsky, R. “Storage Resource Broker - Managing Distributed Data in a Grid,” Computer Society of India Journal, Special Issue on SAN, Vol. 33, No. 4, 2003, pp. 42-54.

39 Romberg, M. “The UNICORE Architecture–Seamless Access to Distributed Resources”, Proceedings of the Eight IEEE International Symposium on High Performance Computing, Redondo Beach, CA, USA, August 1999, Pages 287-293.
40 Southern California Earthquake Center’s Community Modeling. Environment, http://www.scec.org/cme/.
41 Stallman, R. Debugging with GDB – The GNU Source Level Debugger, Edition 4.12, Free Software Foundation, January 1994.
42 Stevens, R. Tipney, H.J. Wroe, C. Oinn, T. Senger, M. Lord, P. Goble, C.A. Brass A. and Tassabehji M. “Exploring Williams-Beuren Syndrome Using myGrid”, Proceedings of 12th International Conference on Intelligent Systems in Molecular Biology, 31st Jul-4th Aug 2004, Glasgow, UK, published Bioinformatics Vol. 20 Suppl. 1 2004, pp 303-310.
43 Sudholt, W., Baldridge, K., Abramson, D., Enticott, C. and Garic, S. “Parameter Scan of an Effective Group Difference Pseudopotential Using Grid Computing”, New Generation Computing 22 (2004) 125-135.
44 Sudholt, W., Baldridge, K., Abramson, D., Enticott, C. and Garic, S., “Application of Grid computing to parameter sweeps and optimizations in molecular modeling”, Future Generation Computer Systems, 21 (2005), 27-35.
45 Tan, J, Abramson, D. and Enticott, C. “Bridging Organizational Network Boundaries on the Grid”, IEEE Grid 2005, Seattle, Nov 2005.
46 Tanaka, Y. Takemiya, H. Nakada, H. and Sekiguchi, S. “Design, implementation and performance evaluation of GridRPC programming middleware for a large-scale computational grid,” Fifth IEEE/ACS International Workshop on Grid Computing, pp. 298–305, 2005.
47 Taylor, I., Wang, I., Shields, M. and Majithia, S. “Distributed computing with Triana on the Grid” In Concurrency and Computation:Practice and Experience , 17(1-18), 2005.
48 von Laszewski, G. Alunkal, B. Amin, K. Hampton, S and Nijsure, S. GridAnt-Client-side Workflow Management with Ant, 2002, http://wwwunix.globus.org/cog/projects/gridant/
49 Watson, G. and Abramson, D. “The Architecture of a Parallel Relative Debugger”, 13th International Conference on Parallel and Distributed Computing Systems - PDCS 2000, August 8 - 10, 2000.
50 Yarrow, M. McCann, K. Biswas, R. and Van der Wijngaart, R. “An Advanced User Interface Approach for Complex Parameter Study Process Specification on the Information Power Grid”, Proceedings of the 1st Workshop on Grid Computing (GRID 2002), Bangalore, India, Dec. 2000.

Chapter 2: Nimrod and Parametric Computing



The papers in this chapter discuss the early work on parametric computing, and its implementation in a tool called Nimrod. Papers (2), (3) and (4) discuss the evolution of Nimrod from a dedicated tool to a general-purpose parametric computing environment for cluster based supercomputers. Paper (5) shows how this was extended to serve networks of clusters. Paper (6) discusses a commercial implementation of parametric computing on Linux clusters.

(1) Abramson , D., Sosic , R., Giddy , J., Cope , M. "The Laboratory Bench: Distributed Computing for Parametised Simulations", 1994 Parallel Computing and Transputers Conference, Wollongong, Nov 94, pp 17 – 27. 2-1 .. 2-11

(2) Abramson, D.A., Cope, M. and McKenzie, R. "Modelling Photochemical Pollution using Parallel and Distributed Computing Platforms", Proceedings of PARLE-94, pp 478 - 489, Athens, Greece, July 1994. 2-12 .. 2-24

(3) Abramson D., Sosic R., Giddy J. and Hall B., "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995, pp 112-121. 2-25 .. 2-34

(4) Abramson, D., Barak, A and Enticott, C. "Job Management in Grids of MOSIX Clusters", 16th International Conference on Parallel and Distributed Computing Systems, pp 36 – 42, August 13 - 15, 2003 Reno, Nevada, USA. 2-35 .. 2-41

(5) Abramson, D. "Parametric Modelling: Killer Apps for Linux Clusters", The Linux Journal, #73, May 2000, pp 84 – 91. 2-42 .. 2-46

Contributions

I was the major contributor and lead author on all of these papers. Sosic was a colleague at Griffith University; Cope provided expertise in air quality management; Hall was a PhD student (not mine) who provided a case study in laser physics; Barak was a colleague at the Hebrew University in Israel who provided expertise on MOSIX; Giddy, was a research associate under my supervision.

The Laboratory Bench: Distributed Computing for Parametised Simulations

D. Abramson †
R. Sosic †
J. Giddy ‡
M. Cope §

†School of Computing and Information Technology
Griffith University
Kessels Rd, Brisbane,
Queensland, 4111
{davida, sosic}@cit.gu.edu.au
Phone: +61-7-875 5049
Fax: +61-7-875 5051

‡ Co-operative Research Centre for
Distributed Systems Technology
jon@cit.gu.edu.au

§ Victorian Environment Protection Authority
546 Collins St,
Melbourne, 3001
mcope@tuan.cse.rmit.edu.au

Abstract: This paper concerns the use of distributed computers for solving large scientific modelling problems. In many fields, a designer may wish to explore a set of alternative scenarios. If numerical simulation is used as the experimental process, then this means executing a number of independent jobs and then aggregating the results in some way. There are currently two main ways of organising the execution on multiple computers, namely the use of remote job execution systems, or building distributed applications. This paper explores both of these, and proposes a user interface based on a *laboratory bench* metaphor. It then describes a prototype system with the advantages of both current job generation methods. The system is built on top of the Distributed Computing Environment from the Open Software Foundation. A real world example, photo chemical pollution modelling, is used as a sample application.

1994 Parallel Computing and Transputers Conference, Wollongong, Nov 94, pp 17 27.

1. INTRODUCTION

High performance computers are being used to allow scientists and engineers to simulate complex systems. Simulation allows *many* different trials to be conducted in the time required for *one* real laboratory experiment, thus increasing the productivity of the researchers. Further, it allows experiments which are either too dangerous or expensive to be simulated using advanced mathematical models. There are many examples of these systems being used routinely in industry, such as the design of aircraft, automobiles, buildings and mine sites. Also, such models are being used for performing environmental impact statements covering a range of scenarios.

Such computational simulations require very high levels of performance if they are to provide meaningful results in a timely manner. To date, supercomputers have been employed to accelerate the execution of any one simulation run; if multiple simulations are

required to evaluate a range of options then they tend to be run sequentially one after another. However, the proliferation of high powered workstations has allowed users to execute multiple independent simulations concurrently, thus reducing the time through parallel execution. This scheme is much cheaper than using supercomputers, however, the management of such jobs can be tedious. Consequently, a number of remote execution systems have been developed for queuing jobs to remote workstations. Even given the availability of such queuing systems, the management of massively parallel distributed computations is difficult.

A further disadvantage of the current queuing systems is that they do not provide a user interface which is related to the application code, but rather, they provide a view of the computing network and its resources. In this paper we discuss the concept of providing the user with an abstraction of a *laboratory bench* on which the experiments are performed, and the underlying computational resource is displayed in terms of the experiment. This model differs from current systems which are technology oriented rather than user oriented.

In line with the dramatic increase in the number of networked workstations, there has been a rise in *open distributed processing* technology. There are now a number of commercial systems for building distributed applications, i.e. applications that execute across many machines. However, to date, there are very few simulation programs which make use of this level of technology because it requires major restructuring of existing codes.

In this paper we discuss the functionality of existing systems that provide remote job execution. We also discuss the functionality of *open distributed systems* and propose a set of extensions to current remote execution packages which bridge the gap between remote job execution and open distributed processing. We then discuss how a distributed computer can be effectively used in engineering design. Finally, we illustrate the power of such a hybrid system through its application to a study of various air pollution scenarios.

2. PARAMETISED ENGINEERING COMPUTATIONS

A wide range of scientific and engineering experiments can be solved using numeric simulation. Examples include finite element analysis, computational fluid dynamics, electromagnetic and electronic simulation, pollution transport, granular flow and digital logic simulation. Accordingly, some very large codes have been written over the years, mostly in FORTRAN and mostly with primitive user interfaces. A typical FORTRAN simulation program may consist of 50,000 lines of code and usually performs its input and output by reading and writing files. This approach has been more than adequate for large mainframes, vector supercomputers and even modern workstations. Users typically generate a set of test input files and submit the jobs to a batch queue for execution. Post processing is used to display and visualise the results.

For many design processes the user wishes to study the behaviour of some of the output variables against a range of different input scenarios. The following brief list gives an indication of the types of parametric studies that might be performed:

- An architect may wish to study the effect of varying the rigidity of a particular beam in a building design and produce a solution which optimises both cost and safety;
- An aerospace engineer may wish to alter the shape of an aerofoil and observe the effect on drag, lift and cost;
- An electronics designer may wish to vary component tolerances and observe the effect on the performance of some analogue circuit;
- An aerial designer may wish to alter the shape of an antenna and observe its gain.
- An environmental engineer may wish to perform an impact study on varying the amount of waste sent into a dump site.

Using existing programs to perform such studies is extremely time consuming and can involve very complex and long job initialisation phases. Typically, the user must create a set of input files for each of the different parameter settings, run the modelling program against each set of files, collate and merge the output files and finally produce some form of condensed output. For example, the output may simply be a low dimensional plot (often two or three dimensions) against the input parameters. For any one run, Pancake [9] has measured that users spend in the order of 10% of time performing job setup; for a multiple scenario experiment this may consume even more time because the operation of parametisation and control are more complex.

If the job set up is performed manually, great care must be taken to ensure that the input and output files are kept correlated. There may be insufficient disk space to store all of the input files on one file system, so they may need to be generated incrementally. Finally, some of these modelling programs take hours to execute to produce one point in the output visualisation; if tens or hundreds of runs are required it may not be possible to perform the experiment in a reasonable time. Further, it is often not possible to tie up an expensive supercomputer for the necessary time, making accurate simulations infeasible. In the next section we will discuss the technologies which are available for providing the necessary computational resource, and how the parametric experiments can be managed.

3. DISTRIBUTED SUPER COMPUTERS

Traditional super computers achieve their performance either through vector hardware or tightly coupled multiprocessors. Providing a program exhibits a high degree of parallel or vector activity, the performance of these systems can be large. However, they are expensive compared to workstations because they employ special hardware and/or software. *Distributed Super Computing* refers to the use of a large number of loosely coupled machines to perform one task. Because the speed of the interconnection network between workstations is slow compared that those in tightly coupled parallel machines, they cannot be used for general parallel computing unless the tasks which are executed are allocated in very large units.

Until recently there have been two main ways of using distributed supercomputers. One, through remote job execution of multiple jobs, and the other through genuinely distributed applications.

3.1 Remote Job Execution

There are now a number of software packages which help manage queues of work to remote computer systems. Some examples are Condor, DQS, DJM, LoadLeveler, LoadBalancer, LSF, CODINE and NQS/Exec [2, 3, 4, 5, 6, 7, 8, 10]. Whilst Condor, DQS and DJM are in the public domain, the others are commercial products. They all provide the basic service which is summarised in Figure 1. Users generate a number of *jobs* and then submit them to the queue management system. The jobs are run on available machines and the results returned to the controlling machine. Most of the systems allows users to access files on the originating machines through networked file systems such as NFS and AFS. Some of the systems provide graphical user interfaces to help them trace their jobs as they move through the system.

An important feature of this type of distributed supercomputing is that the jobs are unaware that they have been executed in a distributed manner. This has the clear advantage that the programmer need only generate a sequential task, and the replication occurs through the use of multiple jobs with different input data. The disadvantage is that users must concern themselves with the task of generating multiple jobs. This process can be complex and error prone, especially when some of the jobs fail to execute because they have terminated early. Unless special software in written to generate the jobs, the user cannot monitor the progress of the jobs in a way which is meaningfully related to the original parametised request. For example, the user requires the computation of a number of different data sets, but the remote job execution system deals in terms of jobs and remote machines.

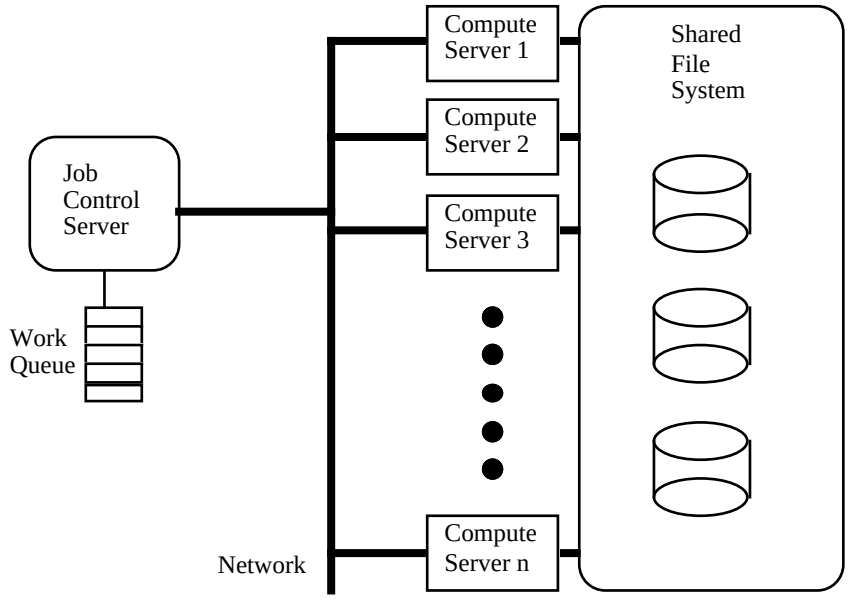


Figure 1 - a generic remote job execution framework

3.2 Distributed Applications

Distributed applications are ones in which the *programmer* is aware of the distributed nature of the computational platform when the program is written. One of the most common mechanisms provided for writing distributed computations is the *remote procedure call*. A remote procedure call has similar semantics to a conventional procedure call, except that the execution of the procedure occurs on a computer different from the caller. Parameters are passed into the called procedure and results are returned when the procedure terminates using a copy in-copy out routine. Figure 2 shows an example of a distributed application. It shows a program running on Machine 1 which acts as a *client*, and which requests services of 3 *servers* through remote procedure calls. The code on Machine 2 acts as a server for the machine 1 client as well as a client of servers running on machines 3 & 4.

This approach has the advantage that the user interface can be tailored to take advantage of the distributed mode of computation, and the metaphor of a *laboratory bench* can be provided. In this scheme, users enter the important parameters for the experiment. The work is automatically generated, distributed and the results are gathered and presented. This metaphor will be discussed further in Section 4.

A number of commercial systems are available for performing remote procedure calls within conventional programs. The Distributed Computing Environment (DCE) from the Open Software Foundation (OSF) is an open interface which is being supported by a number of computer vendors [11]. DCE allows a user to write a program which performs its work on more than one processor by executing a number of concurrent remote procedure calls. Since the calls are embedded in existing sequential languages, and since procedure calls block until the called code completes, a *threads* package is supplied with DCE. This allows a program to fork into a number of concurrent lightweight processes, each of which performs a remote procedure call and waits for the results. This arrangement is summarised in Figure 3.

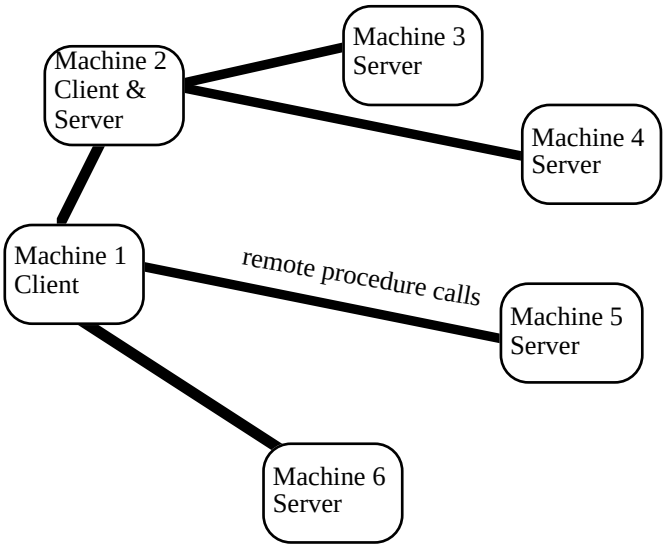


Figure 2 - sample distributed application

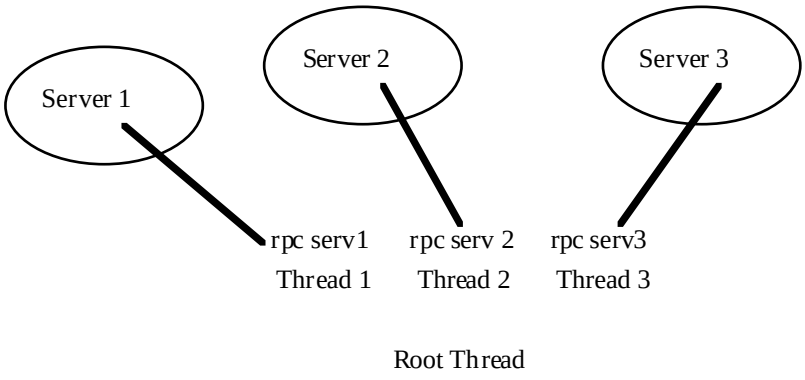


Figure 3 - Executing multiple concurrent RPCs within a distributed application.

The DCE approach is very attractive because it makes it possible to hide the underlying concurrency from the user whilst taking advantage of many idle workstations. However, it requires significant changes to the application code, which may be difficult or impossible. Many existing engineering applications are distributed as executable binaries and cannot be modified. Another disadvantage of DCE is that it has been designed to interface with C. Most high performance scientific and engineering codes are written in FORTRAN, and it would be very difficult to make use of DCE Threads and RPCs from within a FORTRAN program.

4. A NEW DISTRIBUTED REMOTE JOB CONTROLLER

4.1 A Laboratory Bench Metaphor

As discussed in the previous section, there are two main ways of using distributed supercomputers for parametric modelling experiments. The remote job execution systems have the advantage that they do not require any changes to the modelling programs. However, they provide a very technically oriented user interface and are structured in terms of the computational resources rather than the modelling exercise. Further, the user is responsible for setting up the appropriate directories with the files for each job and managing the distribution to machines, which can be a laborious and error prone process.

On the other hand, using distributed computing systems like DCE allow the user program to present a view of the experiment which matches the overall requirements. Further, the code which is necessary to access the computational resources is hidden in the program. The application is responsible for managing the addition of new resources and withdrawal of others, and this can, to a large extent, be hidden from the user. However, it has a major disadvantage that the application code must be modified to incorporate the DCE constructs. Given the large base of existing scientific code, the cost of modifications might be prohibitive.

To provide a better environment for parametric simulations, we are currently developing a user interface and underlying modelling system which presents the user with a *laboratory bench* metaphor. This metaphor provides the advantages of both the remote queuing system and distributed computing environments. In this scheme, as illustrated in Figure 4, jobs are first prepared, processed and then the results are aggregated and visualised for interpretation. The computational resources required to perform the experiment are hidden from the user, who is responsible only for specifying the input and output requirements.

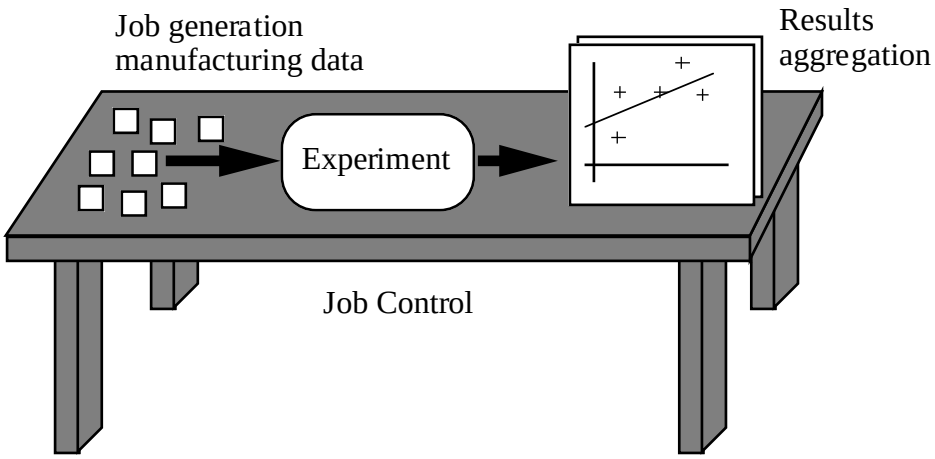


Figure 4 - A metaphor for a laboratory bench

The bench is built in a way which allows it to take advantage of distributed systems like DCE, whilst not requiring modification to the application code. It contains three separate functions, namely, job generation, job control and result aggregation. Job generation and job control make use of graphical user interfaces to provide the user with an application view of the experiment, and use DCE to perform the work distribution. However, the application code is treated as a stand alone program executed via a *remote execution server*. Result aggregation has the knowledge of the distributed nature of the computation, but is separate from the original application code.

4.2 Job Generation

Jobs are generated as a by-product of the different parameter settings. The cross product of all legal parameter values is generated by the system, and each element of the resulting set is processed as an independent job. Figure 5 shows a job generation screen from the tool that we are constructing, containing examples of the different parameters. In this example, the simulation has been designed to model the process of photochemical pollution generation. The screen contains parameters which control the choice of model, the emissions inventory data base, and the number of simulations to run. The latter is controlled by specifying that a number of simulations are required for different levels of the species in the emissions inventory, in this case Nitrous oxides (NOX) and Reactive Organic Compounds (ROCs). This is discussed further in Section 5.

Most modelling applications make use of files for input and output of information. The job generation phase is responsible for sending the input files to the target computer, and

arranging for execution of the code via the job control mechanism. It does this via a remote file transfer server on the target system. Further, it launches the application with the particular parameter settings which correspond to the job, thus the user is free of the responsibility of guaranteeing that the correct files are available for the particular instance of a job.

To test and verify the laboratory bench metaphor, we have built a prototype job generation interface for performing parametric studies of photochemical pollution modelling. We are currently building a generic tool which will generate jobs together with a user designed graphical user interface.

4.3 Job Control

If the jobs are to be distributed to multiple work stations or supercomputers concurrently, then a queuing system must be capable of generating jobs, possibly created on demand to save disk space. As discussed in section 2.1 many queuing systems exist. However, the current systems suffer from two disadvantages:

- They do not use an open distributed processing framework, and are mostly designed for UNIX based operating systems

- They are not integrated into a laboratory bench metaphor, thus users are too concerned with the operational aspects of the computational experiment, such as job generation and result display.

- Jobs are tracked by a job identifier which may have no relationship to the various parameters

We have built a job queuing system which is based around a DCE client for dispatching jobs and a DCE remote execution server for each available machine. Because the DCE specific information is contained in this module, the application can execute as though it is a batch process, and is unaware of its exact location.

Figure 5 shows a sample screen which displays the status of each of the jobs generated. Global statistics, such as the number of jobs which are completed is shown, as well as the status of each job. A job is either waiting for execution, running or complete, as indicated by different icons. In this example, the display is organised so that the two parameters are represented along the X and Y axes of the display, thus it is possible to see which simulations are complete. By clicking on individual job icons, more information can be displayed.

Different job selection algorithms are available, so it is possible to either execute jobs in the order of generation, or in a random order. The latter is useful because it allows aggregation of results to be performed before all jobs have completed, and a partial result can be displayed. In the example considered in this paper, this has the advantage that some preliminary results can be displayed without waiting for all of the jobs to complete. Accordingly, the user may terminate the remaining jobs and begin another experiment.

4.4 Result aggregation

When all of the simulations are completed, the results must be presented in an appropriate manner. The output format for any particular experiment varies depending on the results and how they are to be interpreted, and thus a large amount of flexibility is required in the way that results are presented. For example, in the pollution modelling work we have performed, a simple two dimensional contour diagram is sufficient. However, for some experiments, a three dimensional colour visualisation may be the best way of presenting the results. Accordingly, the framework we have built allows the user to supply a script which is executed, and which displays the data in an arbitrary way.

5. PHOTO-CHEMICAL POLLUTION MODELLING - AN EXAMPLE

This section concerns the distribution of some large scientific modelling programs which are used to compute the transport and production of photochemical smog within an urban airshed. The programs allow scientists in the Victorian Environment Protection Authority of Australia to simulate the smog production in Melbourne and to experiment with pollution reduction strategies. The work was conducted in 1992 as part of a collaboration between the Victorian EPA, the CSIRO Division of Information Technology and the Royal Melbourne Institute of Technology, and serves as an illustration of the power of the laboratory bench metaphor described in this paper.

Simulation has major advantages over direct physical experimentation. It is possible to study many more scenarios than would be physically possible, and it is also possible to measure and assess the effect of control strategies without the enormous expense of implementing them.

Photochemical smog modelling poses some interesting challenges. Such models consume enormous amounts of processor time and memory, and must often be performed within strict time constraints and budgets. Parallel and distributed computing technology has the potential to provide realistically priced platforms for performing such experiments.

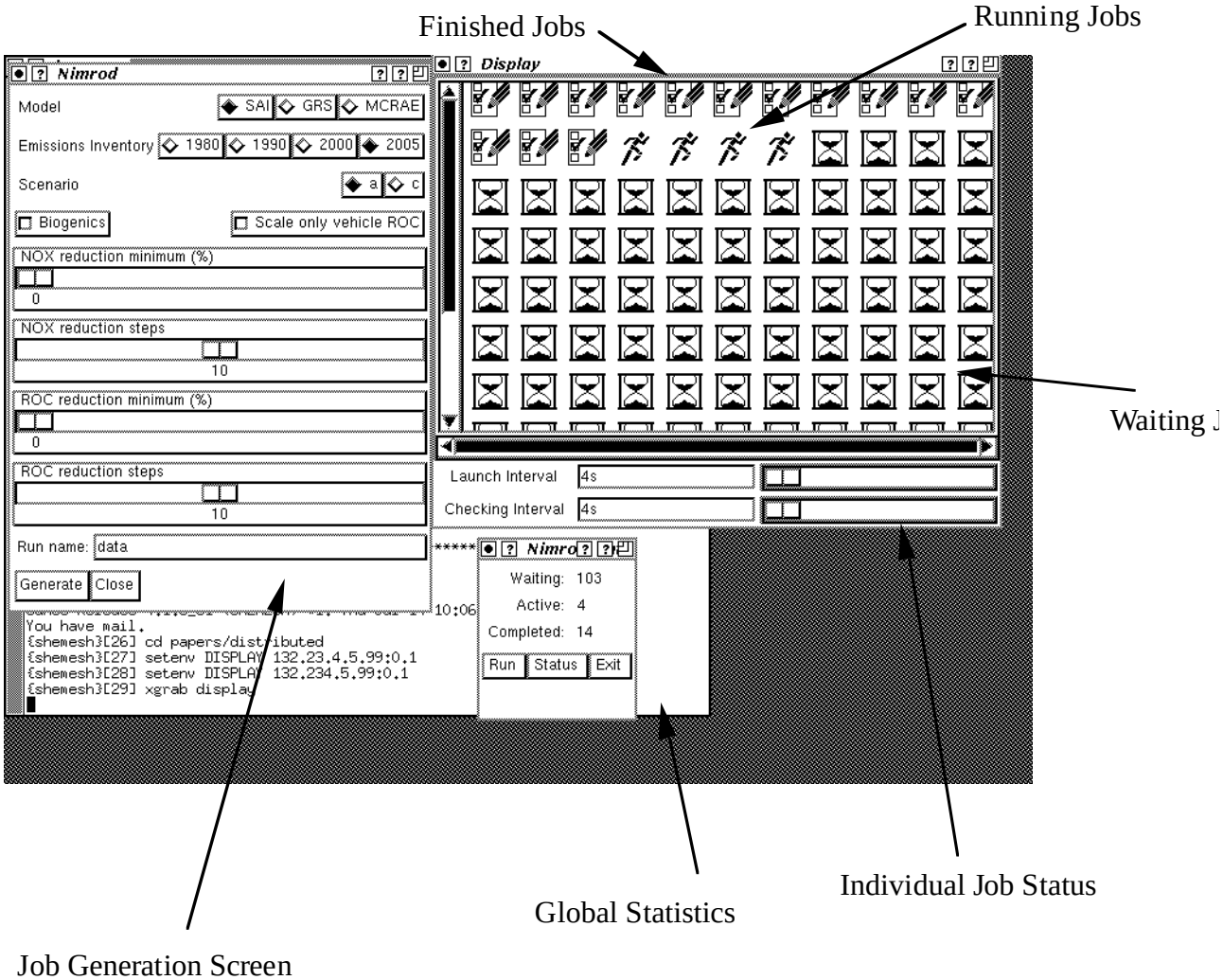


Figure 5 - Sample generation screen and status screens

One of the major uses of photochemical airshed models is to compute oxidant concentrations. Oxidants, such as ozone, are generated as a result of the chemical interaction between various precursors such as oxides of nitrogen (NOx) and other reactive organics (ROCs) in the presence of ultra-violet radiation. Ozone is of particular importance because of its health related side effects; in Melbourne, Australia, the peak ozone levels have been observed to exceed 0.12 ppm in recent years, which is a widely adopted health standard level.

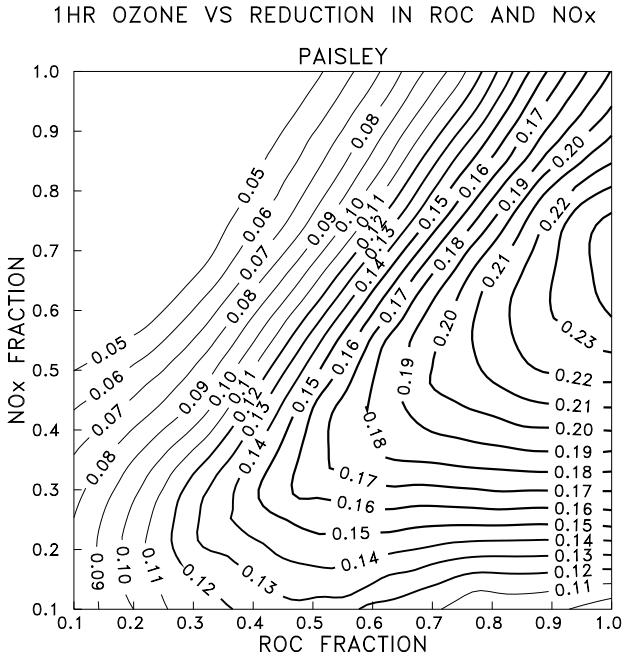


Figure 6 - Sample ozone contours

One way in which the model is used is to determine the sensitivity of the photo chemistry to various input parameters. For example, it is common to plot the peak ozone concentration as a function of NOx and ROC control, as shown in Figure 6. These diagrams then allow the modeller to determine the parameters that most affect the ozone level. The experiments are performed for a number of distinct physical locations. Unfortunately, the amount of work that is required to perform any one contour chart is enormous; if a single scenario takes about 8 hours of workstation time, then one contour chart of 50 independent runs requires 400 hours to perform. If the charts are prepared for up to 10 different weather conditions and emissions data bases, then 4000 hours of compute time are required. Thus, generation of the contour charts requires access to supercomputing technology in order to produce timely results.

Rather than using conventional supercomputers to perform this work, we used a large collection of workstations, physically distributed between Melbourne and Canberra. This provided about 30 Sun Sparc Stations, and utilised machines which would otherwise have been idle overnight. Thus, the work was performed without disturbing the owners of the workstations. Using this resource, it was possible to produce one ozone contour chart like the one shown in Figure 6 overnight.

The project described in this section developed an early version of the laboratory bench system described in this paper. This system contained a user interface similar to the one shown in Figure 5, but made use of UNIX remote execution commands rather than DCE. The study indicated that the laboratory bench metaphor was extremely valuable, and allowed the EPA to evaluate a number of alternative pollution reduction strategies. The success of the project lead to the concept of a generic tool which could be applied to many

different modelling applications. More details of the photo chemical pollution work can be found in [1].

6. CONCLUSIONS

Distributed computers have the potential to provide an enormous computational resource, which can be used for performing parametised numerical simulation experiments. However, the resource must be presented and managed in a way which relates to the original problem if it is to be used effectively by non-experts. In this paper we have discussed the state of the art in distribution of such models, and have proposed an extension to existing job distribution systems.

The laboratory bench metaphor proposed in this paper has been developed using the Distributed Computing Environment. This is an open interface for writing distributed systems. Consequently, the system can make use of a wide range of platforms transparently. The system utilises both of the current methods of performing distributed supercomputing, and thus gains the advantages of both schemes.

We are presently using the system as part of a photochemical pollution modelling study and thus the job generation facility is tailored for this application. The next phase of the project will develop a generic interface which will allow a wide range of engineering applications to be executed.

Additonal issues such as security, checkpointing and job migration will be addressed in future research.

7. Acknowledgments

This work has been performed at Griffith University as part of the Co-operative Research Centre for Distributed Systems Technology (DSTC). The authors which to acknowledge the support of Mr Melfyn Lloyd for proof reading a draft of this paper. We also acknowledge the contribution of Rowen McKenzie of RMIT for his work on a version of the laboratory bench system.

8. References

[1] Abramson, D.A., Cope, M. and McKenzie, R. "Modelling Photochemical Pollution using Parallel and Distributed Computing Platforms", Proceedings of PARLE-94, pp 478 - 489, Athens, Greece, July 1994

[2] Bricker, A., Litzkow, M., Livny, M., "Condor Technical Summary", University of Wisconsin - Madison, October 1991.

[3] Ferstl, F, "CODINE Technical Overview", Genias, April 1993.

[4] Green, T. P., Snyder, J. "DQS, A Distributed Queuing System", Florida State University, March 1993.

[5] International Business Machines Corporation, "IBM LoadLeveler: User's Guide", Kingston, NY, March 1993.

[6] Kaplan, J., Nelson, M. , "A Comparison of Queueing, Cluster and Distributed Computing Systems", NASA Technical Memorandum 109025.

[7] Litzkow, M., Livny, M. and Mutka, M. "Condor - A Hunter of Idle Workstations". Proceedings of the 8th International Conference on Distributed Computing Systems. San Jose, Calif. June 1988.

[8] Litzkow, M., Livny, M., "Experiences With The Condor Distributed Batch System", Proceedings of the IEEE Workshop on Experimental Distributed Systems, Huntsville, AL, October, 1990.

[9] Pancake, C., Cook, C., "What Users Need in Parallel Tool Support: Survey Results and Analysis", Proceedings of 1994 Scalable High-Performance Computing Conference, May 23-25, Knoxville, Tennessee.

[10] Revor, L. S., "DQS Users Guide", Argonne National Laboratory, September 1992.

[11] Rosenberry, W., Kenney, D., Fisher, G. "Understanding DCE.", O'Reilly & Associates, Inc. Sebastopol, California. 1992.

Modelling Photochemical Pollution using Parallel and Distributed Computing Platforms

D. Abramson †
M. Cope §
R. McKenzie ‡

† School of Computing and Information Technology
Griffith University
Kessels Rd, Brisbane,
Queensland, 4111
davida@cit.gu.edu.au
Phone: +61-7-875 5049
Fax: +61-7-875 5051

§ Victorian Environment Protection Authority
546 Collins St,
Melbourne, 3001

‡ Department of Computer Systems Engineering
Royal Melbourne Institute of Technology
P. O. Box 2476V,
Melbourne, 3001

Proceedings of PARLE-94, pp 478 - 489, Athens, Greece, July 1994

Abstract. Scientists are increasingly turning to numerical simulation in order to investigate and model complex environmental systems. Numerical simulation has enormous advantages over laboratory or field experimentation because it has the potential to allow a much greater parameter space to be considered. Numerical simulation is being used for simulating the formation of photochemical air pollution (smog) in industrialised cities. However, computational hardware demands can be great. In this paper we discuss the computational resources that are required in order to achieve a realistic number of results in a timely manner. We describe the parallelisation and distribution of programs that have been used as part of an air pollution study being conducted in Melbourne, Australia. We also describe some computer tools which proved effective in managing this study.

Keywords: Application of Parallel Systems, Scientific Computing, Design of Parallel Programs, Shared, Clustered and distributed machines.

1. Introduction

This paper concerns the parallelisation and distribution of some large scientific modelling programs which are used to compute the transport and production of photochemical smog within an urban airshed. The programs allow scientists in the Victorian Environment Protection Authority of Australia to simulate the smog production in Melbourne and to experiment with pollution reduction strategies. Simulation has major advantages over direct physical experimentation. It is possible to study many more scenarios than would be physically possible, and it is also possible to measure and assess the effect of control strategies without the enormous expense of implementing them.

Photochemical smog modelling poses some interesting challenges for computer architects. Such models consume enormous amounts of processor time and memory, and must often be performed within strict time constraints and budgets. Parallel and distributed computing technology has the potential to provide realistically priced platforms for performing such experiments.

In this paper we address two different techniques for providing the computational resource necessary to perform real air pollution studies. The system is used as part of the Melbourne Airshed study [1], and is currently being proposed for use in three other studies being performed in Australian cities.

The paper begins with a discussion of the techniques used in modelling photochemical smog, and then describes the computation structure of such calculations. We then describe the hardware infrastructure which was available, followed by the parallelisation and distribution techniques which were adopted for the Melbourne study. One of these activities required the construction of a tool to control the distributed experiments, and this is described in section 5.

2. Modelling Photochemical Pollution

One of the major uses of photochemical airshed models is to compute oxidant concentrations. Oxidants, such as ozone, are generated as a result of the chemical interaction between various precursors such as oxides of nitrogen (NOx) and other reactive organics (ROGs) in the presence of ultra-violet radiation. Ozone is of particular importance because of its health related side effects; in Melbourne, Australia, the peak ozone levels have been observed to exceed 0.12 ppm in recent years, which is a widely adopted health standard level.

2.1 Prognostic Wind Field modelling

Meteorology is an essential component in the formation of photochemical smog. High temperatures and high incident fluxes of radiation are required to drive photochemical smog production, a process which typically takes 4-6 hours. Light winds and strong atmospheric stability are also prerequisites if high precursor and smog concentrations are to be generated. For a city such as Melbourne, these conditions exist during the summer months when synoptic winds are light. However, Melbourne's location, on a coast and surrounded by a bowl of hills to the north, east and west is such that complex meso-scale meteorological flows are usually present during these conditions. Successful simulation of photochemical smog production in turn requires the accurate simulation of these air flows.

Over the years, a number of prognostic mesoscale wind field models have been developed to compute 3-dimensional wind fields [2][3]. The model which was used in this work is based on the Colorado State University (CSU) mesoscale model, which has been shown to have good skill in predicting the wind fields where diabatic heating and cooling and orographic effects predominate. These conditions constitute a large proportion of oxidant event days within the Melbourne vicinity, making the model particularly suitable.

Wind field models attempt to simulate the processes important in the Earth's weather. They are based on the dynamics and thermodynamics of the large scale flow, as described by the Navier Stokes equations. Physical processes, such as the effect of solar and longwave radiation and interactions with the surface must also be included.

The partial differential equations that describe the atmosphere are highly non-linear and cannot be solved directly. Thus, some sort of discretization must be used to obtain

a numerically solvable system. There are many different mathematical schemes for solving such equations, however, these usually result in the solution of a number of finite difference equations. The various variables in the model are discretized over a finite grid, and the value of each variable at every grid point is computed. This computation is highly repetitive and lends itself well to vectorization and parallelisation, as will be discussed later in the paper. The modified form of the CSU code used in this work allows for multiple grid resolutions to be nested within each other, rather than applying one uniform grid to the region being modelled. This has the advantage that less work can be performed on regions of less interest to the study, however, it does complicate the parallelisation of the code.

2.2 Photochemical airshed modelling

There are a number of models capable of simulating the chemical transport and production that occurs in the atmosphere. During this work we experimented with two models. One was the Urban Airshed Model (UAM), which is a three dimensional Eulerian photochemical diffusion model developed by SAI [4]. The other is the CIT model, developed by McRae et al [5] originally at California Institute of Technology. Both perform the same basic computation, although they use different mechanisms for modelling the chemical interactions and different numerical procedures. Two distinct computations are performed. One is computing the concentrations of the various chemical species and the other is advection and diffusion of the species. The computation of the concentrations is performed by solving the following differential equation:

$$\frac{\partial c_i}{\partial t} + \nabla.(uc_i) = R_i.[c_1, c_2, c\ , \ldots, c_n; T, t] + Q_i$$

where $c_i(x,t)$ is the concentration of species, $u(x,t)$ is the advective flow field, $R_i.[c_1, c_2, c\ , \ldots, c_n; T, t]$ is the net rate of chemical production at position $u(x,y,z)$, temperature $T(x,t)$, and time t, and Q_i is the rate of direct emission of pollutant i above ground level. R_i is described by a set of coupled ordinary differential equations. This set of equations can be quite difficult to solve in practice because it is very stiff, and thus very small time steps are required. Consequently, quite a large amount of processor time is devoted to computing the chemical concentrations.

The rate of advection of chemical species *i* by the wind field may be written in two dimensional form as

$$\frac{\partial c_i}{\partial t} = -\frac{\partial uc_i}{\partial x} - \frac{\partial vc_i}{\partial y},$$

where u and v are the horizontal components of the three dimensional wind vector *U = (u,v,w)*.

In order to solve the equations given above, the three dimensional region is broken into a number of finite volumes and a finite difference form of the equations is solved numerically.

An important aspect of the computation of the species concentrations is that it is possible to compute the concentrations independently for each finite volume created

by the domain decomposition. However, information is conveyed between these volumes during diffusion and advection processes.

3. Computational Structure

The photochemical airshed models described in the previous section are used in two different computational experiments. One of these generates the ozone (and other pollutant) distribution for one particular emissions data base and weather scenario, and is used for event description and model verification. The other generates multiple scenarios and aggregates the results into one report. The latter is used to evaluate various control strategies.

3.1 Modelling a Single Scenario

One of the uses for the model is to test various control strategies and perform source sensitivity experiments. In this mode, the modeller interacts with the computer system by changing some important parameters and then performing a simulation. For realistic results a minimum, integration time of two days is required, and this can take up to 16 hours on a typical workstation. Such response times are far too long to be useful in performing source sensitivity experiments and severely limit the usefulness of the system. Typical control strategies may involve shutting down certain industrial sites for specified hours, reducing traffic flow along particular roads, etc. To achieve a turn around of about 1 minute, the model would require a sustained performance in the order of 3 Gflops which cannot be achieved without supercomputing hardware (either vector or parallel).

As part of the Melbourne study, a number of different visualisations were developed to display the results of the computation in a meaningful manner. Figure 1 shows the output screen of one of the visualisations of a single scenario. It shows the ground level concentrations of four different chemical species, Oxides of Nitrogen, Carbon Monoxide, Parafins and Ozone. The discretization of the grid into block shaped regions can be clearly seen. The display shows a false coloured image of the intensity of the species over the Melbourne region. The text on the screens corresponds to the suburbs in which air monitoring stations are installed. As the animation proceeds the pollutant concentrations can be seen to change and move.

This type of interface allows a modeller to experiment with different control strategies and get fast visual feedback on the effectiveness of the strategy. The mode of interaction clearly favours the application of lateral thought because the experiment is visualised in the language of the modeller.

3.2 Generating Ozone Contours - Multiple Scenarios

Another way in which the model is used is to determine the sensitivity of the photo chemistry to various input parameters. For example, it is common to plot the peak ozone concentration as a function of NOx and ROG control. These diagrams then allow the modeller to determine the parameters that most affect the ozone level. The experiments are performed for a number of distinct physical locations. Unlike the single scenario case, these simulations do not require particularly fast response for any one run. However, the amount of work that is required to perform any one contour

chart is enormous; if a single scenario takes about 8 hours of workstation time, then one contour chart of 50 independent runs requires 400 hours to perform. If the charts are prepared for up to 10 different weather conditions and emissions data bases, then 4000 hours of compute time are required. Thus, even the contour charts require access to supercomputing technology in order to produce timely results.

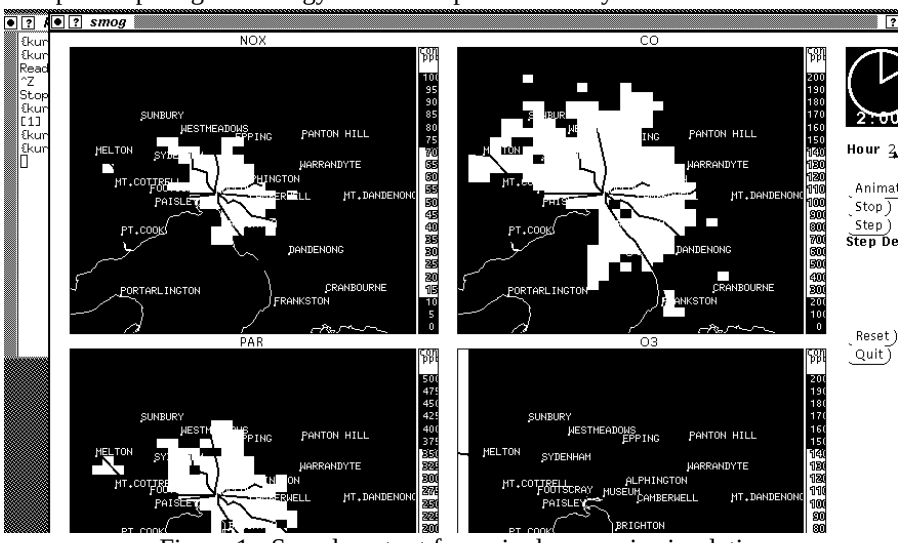


Figure 1 - Sample output from single scenario simulation

Figure 2 shows a sample ozone contour for one location in the Melbourne region. It clearly shows the non-linear effect of varying ROGs and NOx on the ozone concentration. In some regions of the control space, increasing one of the precursors can increase the ozone, and in others it can decrease the ozone. Thus, a simple minded strategy of decreasing the precursors may not have the desired effect of decreasing the oxidants. It is this effect that makes it essential to correctly model the process when evaluating control strategies.

4. Parallel and Distributed Platforms

4.1 Parallelisation or Distribution?

The two uses of the models described in the previous section lend themselves to two different parallelisation techniques and different hardware platforms. In order to generate one pollution scenario quickly it is necessary to deliver sustained super computer performance for the duration of the user interaction, ie. in the order of a minute. Clearly, the only way to deliver this level of performance is through either vector or parallel computing platforms (or a combination of the two). Consequently, the internal algorithms of the model must be vectorised or parallelised. Because one of our major concerns was cost we chose to parallelise the model and run it on a number of small parallel project level supercomputers like the Silicon Graphics Power Series [6]. Whilst this machine cannot deliver the sustained performed mentioned earlier in the paper, the study allowed us to evaluate the effectiveness of parallelising the model. There are now much larger and faster shared memory machines (like the CRAY T3D [7], the Convex SPP [8] and the Kendall Square Research KSR2 [9])

which are capable of delivering the required performance. These machine utilise a distributed virtual shared memory unlike the current shared memory machines.

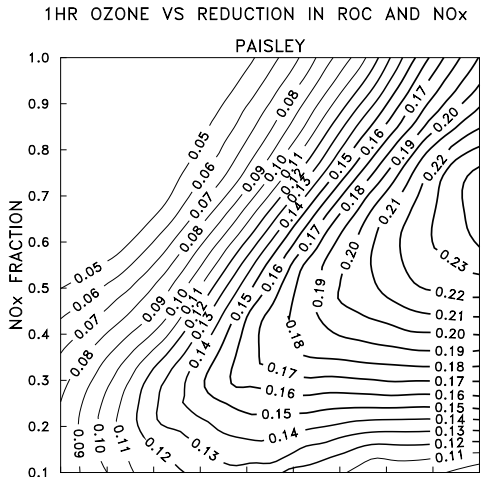


Figure 2 - Sample ozone contours

The generation of ozone contours requires massive computing capacity, but because of the way the output is used the information can be returned in the order of hours. This type of computation is well suited to a distributed platform because each model run is totally independent of every other. Thus, once the initial files have been generated, the runs can be distributed to workstations, and can consume excess unwanted machines cycles. In our case, we distributed the runs across many workstations at different physical sites. One problem with this approach is that the management of the distribution and collection of results can become complex, especially when workstations fail during the computation. To ease this problem we developed some specific tools to assist the process, and these are discussed later in the paper.

4.2 Parallelisation of Airshed Model

The primary aim of our work with the Urban Airshed Model (UAM) and the CalTech Model (CIT) was to determine the suitability of the central algorithms to parallelism on shared memory multiprocessors. Whilst we have parallelised both models, in this section we only present the results from the parallelisation of UAM. The computer used for this investigation was an 8 processor Silicon Graphics SGI 4D/380S.

Parallelising UAM began with identification of program regions which account for significant execution time. By examining time profiling results, the main users of CPU resources were found to be the vertical chemistry (STEP4 and subroutines), vertical advection (STEP3 and subroutines), and x/y integration code (STEP1/STEP2). With the vertical computations accounting for 69% of run time, STEP34 was the first to be parallelised, followed by STEP1 and STEP2 (10% of run time each). As the Power Fortran Analyser (PFA [10]) failed to parallelise the code effectively, the Silicon Graphics mp parallel directives were utilised to provide a

lower level of parallel programming. The main directive used in all cases was the DOACROSS statement which directs the compiler to automatically generate code to run iterations of a Fortran DO loop in parallel. A set of locks provided protection for a few dependent variables inside these DO loops.

The following table indicates the combined effectiveness of the above parallelisation procedures for a single 24 hr simulation. All times are averaged over 5 identical runs. From this it can be seen that a sequential simulation of a 24 hr scenario would have run for over 7 hrs. However with 8 CPUs available it will finish in just over 1 hr.

Threads	CPU secs	Speedup	Ideal	Efficiency (%)
1	25925	1.0	25925	100
2	13270	2.0	12962	98
3	9317	2.8	8642	93
4	7200	3.6	6481	90
5	5881	4.4	5185	88
6	5276	4.9	4321	82
7	4787	5.4	3704	77
8	4091	6.3	3241	79

Where:
CPU secs is the number of CPU seconds used by the master thread
Threads indicates the number of parallel processes chosen
Speedup is the ratio (CPU secs) / (CPU secs for 1 thread)
Ideal is (CPU secs for 1 thread) / (Threads)
Efficiency is (Speedup)/(Threads)*100

4.3. Parallelisation of meteorology

Parallelisation of the prognostic mesoscale windfield model proceeded in a similar manner to the photochemical airshed models. Again the computers used were Silicon Graphics Power series, initially a 4 processor machine and eventually an 8 processor machine. However, unlike the airshed models, in which a substantial proportion of computational time is spent in one section of the program ie the chemical kinetics solver, the computational time was fairly evenly spread across a number of routines. Therefore effective parallelisation of the model required that all routines be modified. Depending upon the structure of the source code, a mixture of coarse grain and fine grain parallelisation was adopted. For example routines which had minimal horizontal coupling such as the surface energy balance and surface temperature calculations, radiation calculations, pressure gradient calculations and vertical diffusion, could be parallelized at subroutine invocation level. On the other hand, horizontal advection, boundary condition routines and housekeeping routines had to be parallelised at the DO Loop level within the relevant routines.

The parallelisation of the code has proceeded in two phases. In the first sweep, an initial speed-up of 3.3 on the 4 processor machine was achieved and was deemed to be acceptable. However after installation on the 8 processor machine, a speed up of only 5.2 was achieved. Even though this indicates that more than 90% of the code was parallelised, this result was considered unacceptable and further optimisation was undertaken. In this second phase, attention was given to the parallelisation of smaller segments of code and also to restructuring the code in order to minimise cache misses.

This was achieve by reversing the indices of arrays for which operations only occurred in vertical columns ie SOIL(I,J,K) was replaced with SOIL(K,I,J) where SOIL contains the vertical temperature profile in the soil as a function of horizontal grid spacing and vertical diffusion is the only process considered. For cases in which arrays stored data which may undergo both horizontal and vertical processing ie horizontal advection and vertical diffusion of the vector wind field, vertical columns of the wind field were loaded into temporary one-dimensional arrays before commencement of the vertical processing. This was found to have a significant effect in cases where the vertical columns were accessed multiple times. A speedup of 6.8 was achieved as a result of these additional modifications.

Phase One				
Threads	CPU secs	Speedup	Ideal	Efficiency (%)
1	7610	1.0	7610	100
2	4013	1.9	3805	95
4	2279	3.3	1903	83
6	1688	4.5	1268	75
8	1471	5.2	951	65
Phase Two				
8	1125	6.8	951	85

Parellelisation of the model across multiple machines is also under investigation. As discussed earlier, the model is generally run as a series of nested grids of increasing resolution in which low resolution information from the outer grids is used to generate boundary conditions for the inner grids. Typically 2-4 levels of nesting are used. In initial attempts at parallelising this procedure have involved running the outer grid on a 4 processor machine, and an inner grid on the 8 processor machine. Boundary condition data for the inner grid is obtained via a shared disk drive. Initial attempts have been very successful however, full operational implementation of this scheme require the development of a dynamic scheduling procedure which ensures that optimum use of the processors are achieved. This becomes even more critical when more than two levels of nested are used and/or the machines are also loaded with other jobs.

4.4 Distribution of Airshed Model

The chemistry model reads input from a number of input files and emits concentration levels to output files. It reads a parameter file which contains the various file names and important parameters for the run. The model was sent to a large number of physically distributed machines, only some of which shared a file system. Thus, we could assume that files which were common across all runs could be simply placed in one shared position in the file system. Conversely, where machines did maintain a shared file system we wanted to take advantage of the fact and avoid duplication of data. In some cases of shared file systems, we were unable to use all of the machines connected to the system because there was insufficient space to create the output files.

Each time the model was exported for execution on another platform the files associated with the run were sent beforehand. We used the Berkeley Unix remote execution commands like *rcp* (remote copy) and *rsh* (remote shell) in order to control

the remote execution [11]. Free use of these commands required the addition of *.rhosts* files on each machine, which proved to be difficult because of security considerations. Consequently, on some machines we were forced to copy a *.rhosts* file to the target system immediately prior to remote job execution using *ftp* [12], and then to delete it after the run had completed. This was performed transparently by placing the commands in the queuing system scripts. Whilst this proved satisfactory it is a poor solution to the problem of accessing secure systems. For example, some secure systems refuse to acknowledge any remote execution commands even if there is a *.rhosts* file. This problem remains an open research issue for distributed computing environments.

5. A Tool for Controlling Distributed Computing

As discussed in the previous section we were able to gain significant computational resource by using workstations in our organisations. These machines, which were normally idle after hours, were used to process the multiple model runs required to generate the ozone contours. Since each contour diagram required in the order of 50 independent runs per chart, managing the process became quite difficult. Because of this, we developed a GUI based tool to control the generation, distribution and display of the results. There are a number of components to the tool.

First, it provides a mechanism for generating a series of model runs based on a set of parameters. In our case, we wanted to generate different simulations based on scaled versions of the emissions data base. Each run simulates a different scenario based on modified values for ROG and NOx levels. Figure 3 shows a sample screen for controlling this generation phase. The user has the choice of selecting which model to use (UAM, CIT or another one called GRS [13]), which emissions data base to select and how to scale the ROG and NOx levels. The levels are scaled by a fixed amount for a number of steps.

The second aspect of our tool is that it distributes the model across a number of workstations connected on the Internet. Figure 3 shows the tool configured to distribute across a number of local workstations; each box shows the status of a named machine. By clicking on a machine more information is produced about the system, and this is shown in Figure 4. The user can set the hours when the workstation is free to accept work. In this way, we were able to make use of machines after hours without disturbing the user of the machine during work hours. Information such as which job is currently being processed by the machine, what the load average is, etc. is also displayed in the host information screen.

The queuing system which underpins the tool distributes work until all of the simulations have been completed. If a machine becomes unavailable during processing it is removed from the set of machines for the remainder of the runs. After each job completes mail is sent to the user about the job. If anomalies have occurred during execution these are reported for user action. An additional load monitor was built so that if a user returned to a machine which was actively running a simulation, the simulation would be suspended until the machine became idle again. Using the queuing system we were able to make use of about 30 workstations, physically distributed between Melbourne and Canberra. Similar features are found in job

distribution systems like Condor [14], DQS [15], CODINE [16] and LoadLeveler [17].

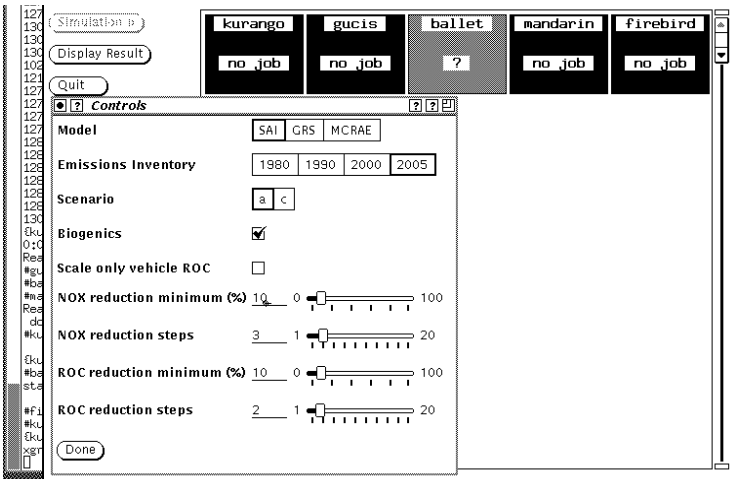


Figure 3 - Sample output of initial startup and status window

The third aspect of the tool is that when the experiment is complete the user can display the results in a condensed form. Figure 5 shows the results of a number of simulation runs. We have extracted one set of ozone readings for one physical site and plotted them against the scaled ROG and NOx levels. The results clearly show the effect of reducing either ROG or NOx on the ground level ozone concentration. Using this information the user may devise some ozone control strategies.

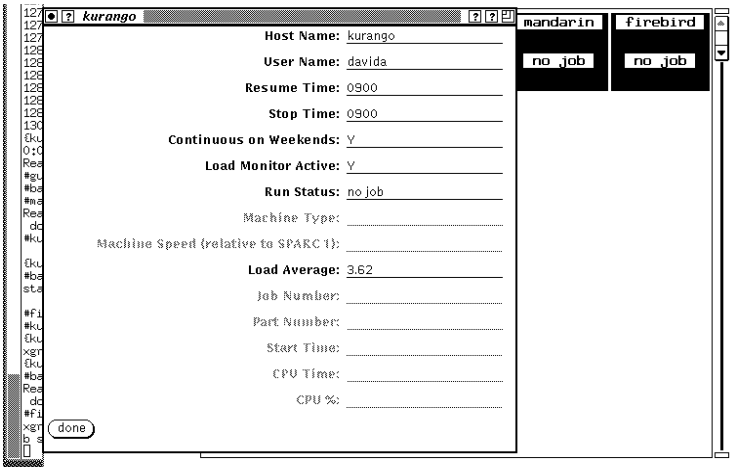


Figure 4 - Host information window

6. Conclusions and future directions

In this paper we have described two different methods for accelerated execution of photochemical smog simulations. One of these concerns the parallelisation of the

models for execution on parallel machines. The other involves the distribution of the model across many, often physically distributed, workstations. The latter form of execution gave us an enormous computational resource at no cost.

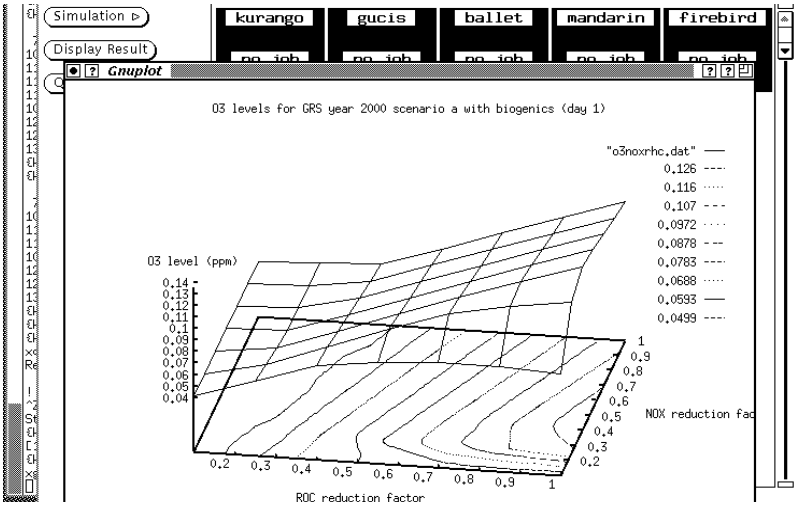


Figure 5 - Display of output from experiment

The parallel forms of the code were only developed for shared memory multiprocessors. This is because most of the available machines were shared memory. There are now a number of much larger, faster, multiprocessors in the Australian research community which rely on message passing code. Accordingly, we are currently parallelising a version of the CIT photochemical airshed model using message passing. Because the code was originally written in Fortran for sequential machines, this operation is proving more difficult than the shared memory port of the same code.

The tool for controlling the distribution was built specifically for controlling the photochemical pollution experiments. Similar parametric studies are also common in other areas of numerical simulation. For example, aeronautical engineers who study the performance of an aerofoil under different conditions may wish to generate a parameterised set of experiments and distribute them across a number of workstations. Consequently, we are developing a more generic tool for generating and controlling such parametric experiments. It will be applicable to arbitrary models which have been written without the knowledge that they will be distributed across multiple platforms. It will be built on top of the standard Distributed Computing Environment (DCE) to aid its use across open systems platforms. This work is being funded under the Co-operative Research Centre for Distributed Systems Technology.

Acknowledgments

This work was performed as part of a collaborative project between the Victorian Environment Protection Authority, the CSIRO Division of Information Technology and the RMIT Department of Computer Systems Engineering. Thanks go to the C.S.I.R.O. Division of Atmospheric Research for providing access to their 8 processor

Silicon Graphics machine on which some of these benchmarks were performed. Thanks go to Paul Pritchard and Binh Pham for reviewing a draft of this paper.

References

[1] Cope M., Carnovale F. and Cook B., 1990. An Air quality Modelling System of Oxidant Transport and Production in Port Phillip Control Region. Proceedings of the 10th International Conference of the Clean Air Society of Australia and New Zealand.

[2] Cope M.E. and Hess G.D., 1993. Meso-scale meteorological modelling for input into a 3-dimensional photochemical oxidant model. Proceedings International Congress on Modelling and Simulation, Perth, Australia.

[3] Physic, W.L. 1988. Review: Mesoscale modelling in complex terrain. Earth-Sci Rev., 25, 199-235

[4] Morris R.E. and Myers T.C., 1990. Users Guide for the Urban Airshed Model: Volume I. EPA Research Triangle Park, N.C. USA (pub. EPA-450/4-90-007A).

[5] McRae G.J., Russell A.G. and Harley R.A., 1992. CIT photochemical airshed model -users manuals, Carnegie Mellon University, Pittsburgh, PA and California Institute of Technology, Pasadena, CA.

[6] Silicon Graphics, 1991. Power Series Performance Guide. Editor Veeleen Roufchaie, Member of Technical Staff Technical Marketing, ASD.

[7] Cray Research, Inc., Cray T3D Technical Summary, September 1993.

[8] Masson, Bob, "Bringing MPP to the Mainstream Customer", White Paper Series, Convex Computer Corporation.

[9] Kendall Square Research, The KSR-1, Technical Summary.

[10] Silicon Graphics, 1991. Power Fortran Accelerator Reference. Document: 007-0629-020.

[11] Comer, D., Internetworking With TCP/IP, Volume 1, Second edition Prentice-Hall 1991

[12] Postel, Jon, and Joyce Reynolds, "File Transfer Protocol (FTP)", Network Working Group Request For Comments - RFC 959, October 1985

[13] Azzi M., Johnson G., Cope M., 1992. An introduction to the Generic Reaction Set photochemical mechanism. Brisbane: Proceedings of the 11th International Conference of the Clean Air Society of Australia and New Zealand.

[14] Litzkow, M., Livny, M. and Mutka, M. Condor - A Hunter of Idle Workstations". Proceedings of the 8th International Conference on Distributed Computing Systems. San Jose, Calif. June 1988.

[15] Green, T. P., Snyder, J. "DQS, A Distributed Queuing System", Florida State University, March 1993.

[16] Ferstl, F, "CODINE Technical Overview", Genias, April 1993.

[17] International Business Machines Corporation, "IBM LoadLeveler: User's Guide", Kingston, NY, March 1993.

The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations

D. Abramson †
R. Sosic †
J. Giddy ‡
B. Hall §

†School of Computing and Information Technology
Griffith University
Kessels Rd, Brisbane,
Queensland, 4111
{davida, sosic}@cit.gu.edu.au
Phone: +61-7-875 5049
Fax: +61-7-875 5051

‡ Co-operative Research Centre for
Distributed Systems Technology
jon@cit.gu.edu.au

§ School of Science,
Griffith University

ABSTRACT

This paper discusses Nimrod, a tool for performing parametised simulations over networks of loosely coupled workstations. Using Nimrod the user interactively generates a parametised experiment. Nimrod then controls the distribution of jobs to machines and the collection of results. A simple graphical user interface which is built for each application allows the user to view the simulation in terms of their problem domain. The current version of Nimrod is implemented above OSF DCE and runs on DEC Alpha and IBM RS6000 workstations (including a 22 node SP2). Two different case studies are discussed as an illustration of the utility of the system.

1 INTRODUCTION

A wide range of scientific and engineering experiments can be solved using numeric simulation. Examples include finite element analysis, computational fluid dynamics, electromagnetic and electronic simulation, pollution transport, granular flow and digital logic simulation. Accordingly, some very large codes have been written over the years, mostly in FORTRAN and mostly with primitive user interfaces. A typical FORTRAN simulation program may consist of 50,000 lines of code and usually performs its input and output by reading and writing files. Users typically generate a set of test input files and submit the jobs to a batch queue for execution. Post processing is used to display and visualise the results. This approach has been adequate for large mainframes, vector supercomputers and even modern workstations.

In parametric studies a range of different simulations are calculated using the same program. Each simulation, which may takes hours to run, computes output variables for one particular set of input conditions. The following brief list gives an indication of the types of parametric studies that might be performed:

- An environmental engineer may wish to perform an impact study on varying the amount of waste sent into a dump site;
- An architect may wish to study the effect of varying the rigidity of a particular beam in a building design and produce a solution which optimises both cost and safety;
- An aerospace engineer may wish to alter the shape of an aerofoil and observe the effect on drag, lift and cost;

- An electronics designer may wish to vary component tolerances and observe the effect on the performance of some analogue circuit;

- An aerial designer may wish to alter the shape of an antenna and observe its gain.

Existing techniques for controlling such studies are extremely time consuming and can involve very complex and long job initialisation phases. Typically, the user must create a set of input files for each of the different parameter settings, run the modelling program against each set of files, collate and merge the output files and finally produce some form of condensed output. For example, the output may simply be a low dimensional plot (often two or three dimensions) against the input parameters. For any one run, Pancake [9] has measured that users spend in the order of 10% of time performing job setup; for a multiple scenario experiment this may consume even more time because the operation of parametisation and control are more complex. If the job set up is performed manually, great care must be taken to ensure that the input and output files are kept correlated.

Parametised simulations can require enormous amounts of processor time. It is not uncommon for such models to require in the order of hours to evaluate any one set of parameters. If only three variables are explored, with 4 values per variable, 64 different simulations must be performed, and this can amount to days of workstation time. Distributed workstations have the potential to provide the necessary computational resource. They are inexpensive, and are often idle for long periods. However, managing the multiple jobs can be complex and time consuming, and thus tools are required in order to harness their power effectively, and to provide a seamless computational platform.

In this paper we discuss a tool, called Nimrod[1], for managing the execution of parametised simulations on distributed workstations. Because each point in the search space is independent, it is possible to execute the tasks concurrently on separate machines. Nimrod is targeted at the application engineer or scientist rather than at a systems programmer. It provides a high level view of the experiment being conducted whilst utilising a wide range of computing platforms to perform the underlying work.

The paper begins with a discussion of distributed supercomputers and surveys the available tools. It then discusses the design and implementation of Nimrod which is built on top of OSF's Distributed Computing Environment (DCE). The paper gives some case studies as examples of the use of Nimrod.

2 DISTRIBUTED SUPER COMPUTERS

Traditional super computers achieve their performance either through vector hardware or tightly coupled multiprocessors. Providing a program exhibits a high degree of parallel or vector activity, the performance of these systems can be high. However, they are expensive compared to workstations because they employ special hardware and/or software.

Distributed Super Computing refers to the use of a large number of loosely coupled machines to perform one task. Because the speed of the interconnection network between workstations is slow compared to those in tightly coupled parallel machines, they cannot be used for general parallel computing unless the tasks to be executed are allocated in very large units. The class of problems being considered in this paper exhibit moderate levels of concurrency (of the order of 50-100 independent model executions) and any one model run may require several hours of compute time. Distributed supercomputers represent an ideal platform for performing such work because the ratio of computation time to job setup time is high, and thus the communication costs of LANs and WANs can be ignored. Further, it is often possible to secure overnight in the order of 100 workstations which would otherwise be idle.

Until recently there have been two main ways of using distributed supercomputers. One, through remote job execution of multiple jobs, and the other through genuinely distributed applications.

2.1 Existing Remote Job Execution Systems

A number of software packages exist which help manage queues of jobs to remote computer systems. Some examples are Condor, DQS, DJM, LoadLeveler, LoadBalancer, LSF, CODINE and NQS/Exec [5, 6, 7, 8, 9, 10, 11, 14]. Whilst Condor, DQS and DJM are in the public domain, the others are commercial products. They all provide the basic service that is summarised in Figure 1. Users generate a number of *jobs* and then submit them to the queue management system. The jobs are run on available machines and the results returned to the controlling machine. Most of the systems allows users to access files on the originating machines through networked file systems such as NFS and AFS. Some of the systems provide graphical user interfaces to help tracing the jobs as they move through the system.

[1] Our work is inspired by "CONDOR: A Hunter of Idle Workstations " [7]. The biblical character Nimrod was another accomplished hunter.

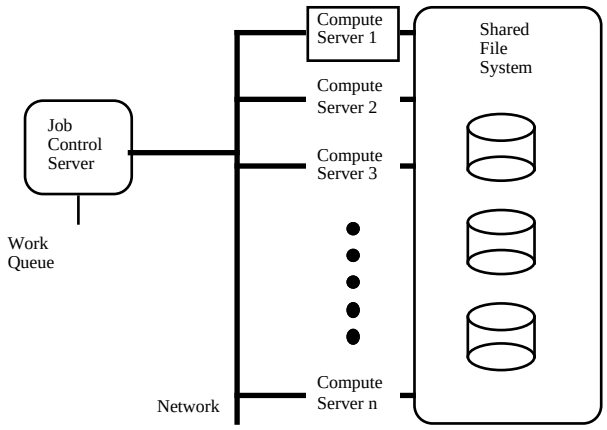


Figure 1 - a generic remote job execution framework

An important feature of this type of distributed supercomputing is that the jobs are unaware that they have been executed in a distributed manner. This has the clear advantage that the programmer need only generate a sequential task. The disadvantage is that users must concern themselves with the task of generating and distributing multiple jobs. This process can be complex and error prone, especially when some of the jobs fail to execute because they have terminated early. Unless special software is written to generate the jobs, the user cannot monitor the progress of the jobs in a way which is meaningfully related to the original parametised request. For example, in parametric strudies, the user requires the computation of a number of different data sets, but the remote job execution system deals in terms of jobs and remote machines.

2.2 Distributed Applications

Distributed applications are built with the knowledge that they will execute on multiple platforms. Thus, an application which performs parametised simulation must be capable of setting up simulations for each of the parameter sets and then executing them using multiples processes. It must handle the initialisation of the processes as well as the aggregation of the results.

There are many tools and environments for building distributed applications. The paradigms vary from remote procedure calls through to co-operating sequential processes. In the remote procedure call system, a program executes a procedure call which causes the routine to execute on another processor. The call looks like a normal procedure call except that parameters are copied in and out rather than passed by reference or value. Since the call blocks until the procedure terminates, multiple calls must be performed in concurrent threads in order to execute more than one simulation at the same time. Examples of remote procedure calls are OSF's DCE [15] system and SUN's RPC [18]. If co-operating sequential processes are used, then the individual simulations can be spawned on other processors, and the parameter information can be distributed through message passing. Unlike RPCs which block, it is possible for the master process to start a number of slaves without waiting for one to complete before starting the next. Examples of portable, multi-lingual, message passing systems include PVM [19] and the P4 system from Argonne National Laboratories [4].

The major advantage of this approach over using a queue manager is that the user interface can be tailored to the problem domain of the application, and all of the details of job generation and resource information can be hidden from the user. Further, the progress of a job through the system can be easily traced in terms of its original parameter settings. Thus, it would be conceivable to build an application which requests a number of parameters from the user, and then automatically generates and distributes the work across a number of workstations. However, the major disadvantage of this approach is that the original application must be modified to take account of distribution. This can be a time consuming task as it may require significant alterations to the program source code. For example, issues such as fault tolerance and restarting of jobs must be addressed from within the application. Further, the source of the application may not even be available to allow the required modifications.

3 NIMROD: A Tool for Parametised Simulations

Nimrod is a tool that combines the advantages of remote queue management systems with those of distributed applications. It provides the user with a problem oriented view of their application without requiring any modifications to their simulation code. Figure 2 shows the overall structure of Nimrod. The user is presented with a generation screen which gives the various parameters to the simulation. After the user specifies the simulation parameters, Nimrod takes the cross product of these parameters and generates a job for each set. It then manages the distribution of the various jobs to machines, and organises the aggregation of results.

By implementing log facilities Nimrod can be restarted at any time throughout an experiment. This attribute is important because a large modelling application may run for days. Over this time various workstations may fail, including the one executing Nimrod. It is also possible to run multiple copies of Nimrod concurrently, allowing simultaneous execution of experiments with different parameter settings, or even different modelling applications.

Rather than assuming a shared file system, Nimrod copies files between systems before and after execution of the program. Thus, when a job is started, the data files are copied to the target system. It is also possible to copy in the executable image for the application, thus it is not necessary to prepare the target system prior to use. When the program terminates, the output files are copied back to the host system.

Most of the Nimrod system is common across all applications. However, the graphical user interface and the control scripts vary depending on the application and parameters. Accordingly, a mechanism is provided for building a new application from a simple description of the parameters and the necessary scripts for running the code. The latter scripts are sufficiently general to make it possible to send and compile the source for the simulation code on the target system.

Figure 4 shows some sample applications of Nimrod, which will be discussed in more detail in Section 5. In this screen dump, Nimrod is being used to control two separate applications. For each application, a generation screen is used to specify the values of parameters. In the laser experiment these are the properties of the model (Non-doppler/Doppler, Isotype1, Isotype2, Linear and Circular), the Detuning range and the Laser Intensity range. In the pollution model these are the model name (SAI, GRS or MCRAE), the emissions inventory (1980, 1990, 2000 or 2005), the scenario (a or c) and the NOX and ROC control steps. The status of all of the pollution jobs can also be seen as well as the status of an individual job. The icons represent the status of the job, such as waiting, running, suspended or finished. The status screen shows 30 jobs out of 403 complete, with 8 currently executing.

Having given an overview of the operation of Nimrod, the next section discusses some of the implementation details.

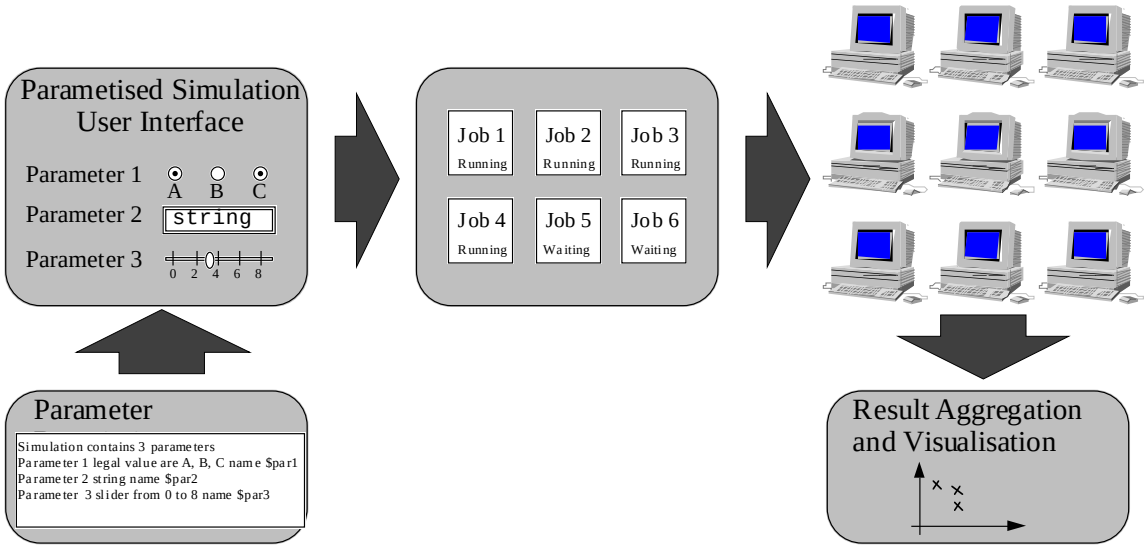


Figure 2 - Overall structure of Nimrod

4 IMPLEMENTATION DETAILS

The structure of the Nimrod implementation is shown in Figure 3. A vertical broken line divides the model into client and server sides. There are two separate servers , one providing file transfer (FT) between the client host and the server host, and the other allowing remote execution (RX) of processes on the server host.

Control of a run begins on the client side at the Job Organisation Tool (JOT). This module provides the user interface, allowing the user to create, monitor, and direct jobs running on many hosts. The Job Distribution Manager (JDM) queues the jobs that are to be run. The JDM interacts with the two servers on each remote host. The client communicates with the servers through the use of user agents. User agents are client side modules working on behalf of remote servers.

When a job is started on a remote host, any required files (such as executables and input files) are transferred to the host. The remote execution server is used to start a shell on the server host and this shell subsequently starts the executable program. On completion, output files are transferred from the remote host to the originating host.

Currently, communication across the network is provided by the Remote Procedure Call (RPC) facility of

DCE. It is possible to replace DCE with another transport mechanism, like PVM by replacing the logic marked in grey in Figure 4.

4.1 File Transfer

As discussed above, before a job can be executed, the remote system must hold a copy of the executable image and any input files. In a parametised run, some of these files may be different for each job, while other files may be the same for all jobs. Nimrod can generate specific data files for each job based in the selection of parameters for that job. There are several potential methods for making files accessible to a remote host. The simplest method is to restrict the use of remote hosts to those that share a file system through a mechanism such as an NFS mount. However, this scheme is restrictive and lacks generality.

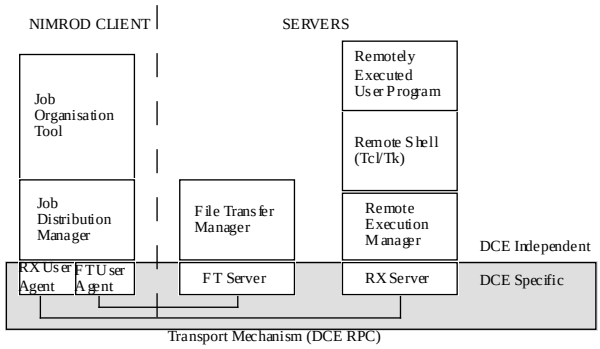


Figure 3 - Structure of Nimrod Client and Servers

A second method involves routing I/O calls (as well as most other system calls) back to the originating host. In this case only the executable must be copied to the remote host. A disadvantage of this scheme is that the overhead in system calls may increase network load unnecessarily. For example, even intermediate temporary files are accessed on the originating host, when they could be handled more efficiently on the remote host. Also, this method requires that the executable is relinked with a customised system library and that the client machine stays up for the entire time the remote job is executing. This latter requirement imposes a serious restriction when jobs run for a long time and when the network is unreliable.

A third method uses a file transfer server and requires explicit copying of the executable and all data files to the remote host. This has general applicability since any host running a file transfer server can accept jobs. In addition, if necessary, an NFS mount can be used for efficiency reasons. Nimrod uses this file transfer method because it allows efficient use of the network and provides resilience to network failures.

The file transfer service consists of three modules. The File Transfer Manager contains the server side functions that perform server activity related to file transfer. The File Transfer Server performs server initialisation and listens for requests. The File Transfer User Agent provides a high-level interface to the functions of the File Transfer Server.

The procedures in the File Transfer Manager are independent of DCE whils the FT Server is DCE dependent. In the current DCE implementation the procedures for transferring files to and from remote hosts make use of DCE pipes for data transfer. DCE pipes provide an efficient way of transferring large amounts of data through RPCs [16]. However, they require the use of callback procedures for placing data on the pipe, for removing data from the pipe, and for allocating buffer space for received data. Every time a file transfer is to be performed, a pipe is created, the callback procedures are registered with the DCE runtime library, and the pipe is passed as a parameter to the RPC. This requires that the code calling the RPC is aware of DCE pipe processing. At the same time, the callback routines used for file transfer are static. They always perform the same basic operation of data transfer between the DCE pipe and a local file.

To reduce the complexity of the Nimrod client, the File Transfer Server interface is abstracted by a File Transfer User Agent which hides the use of DCE as the transport mechanism. DCE binding handles are replaced by an abstract file server handle. The use of pipes in the RPC interface is replaced by a naming scheme for the file's source and destination. This allows applications using the File Transfer server to use other transport mechanisms simply by replacing the FT User Agent and FT Server modules. For example, to make use of a distributed file system, the FT User Agent can be rewritten to call DFS functions instead of issuing RPCs to a FT server.

4.2 Remote Execution

The remote execution (RX) server is responsible for starting a process on a remote host, and is partitioned using the same principles as the FT server. Processes can be started with arbitrary command arguments and execution environments. Terminal I/O can be redirected to a file, obtained through further RPCs, or ignored completely. Processes can be queried for status by clients such as termination and suspension. The server allows forced termination, suspension and continuation of processes providing these functions are supported by the local operating system.

In order to control the load on any given machine, a server only accepts jobs until various thresholds are

exceeded. For example, if the system load is too high then the server may reject execution requests. This scheme caters for uni-processors, on which only one job may be run, and multi-processors, which can accept many jobs concurrently.

The facililities for File Transfer and Remote Execution provide the basic infrastructure required by Nimrod. The remaining modules are associated with the particular tasks of distributing the jobs in an intelligent manner and providing capabilities to monitor and control the jobs.

4.3 Remote Shell

The remote execution server provides some basic information about the status of the job being executed. A remote shell is used to provide more detailed status information to be obtained than that provided by the RX server.

When a new job is started the RX server is used to run a remote shell, which is controlled using calls to the RX server's I/O RPCs. The shell can be used to start the job and wait for the termination of the job. In addition, a customised shell can monitor the intermediate progress of the job. A simple way of doing this is to monitor the size of a particular output file. A more complex, but possibly more useful method is by directly accessing variables in the job's process space. We are currently experimenting with providing this information by using a portable debugging library [17]. Thus, using this scheme, users can observe how far a job has progressed by looking at internal variables of the code, such as time step and loop counters.

4.4 Job Distribution Manager

The Job Distribution Manager (JDM) performs a similar task to that provided by existing systems such as Condor and LSF. Given a set of jobs, it runs them on under-utilised hosts on the network, and provides basic status monitoring (e.g. job is running / suspended / terminated normally or abnormally).

The task of distributing the jobs can be broken down into a number of subtasks. For each job, the subtasks to be performed are:

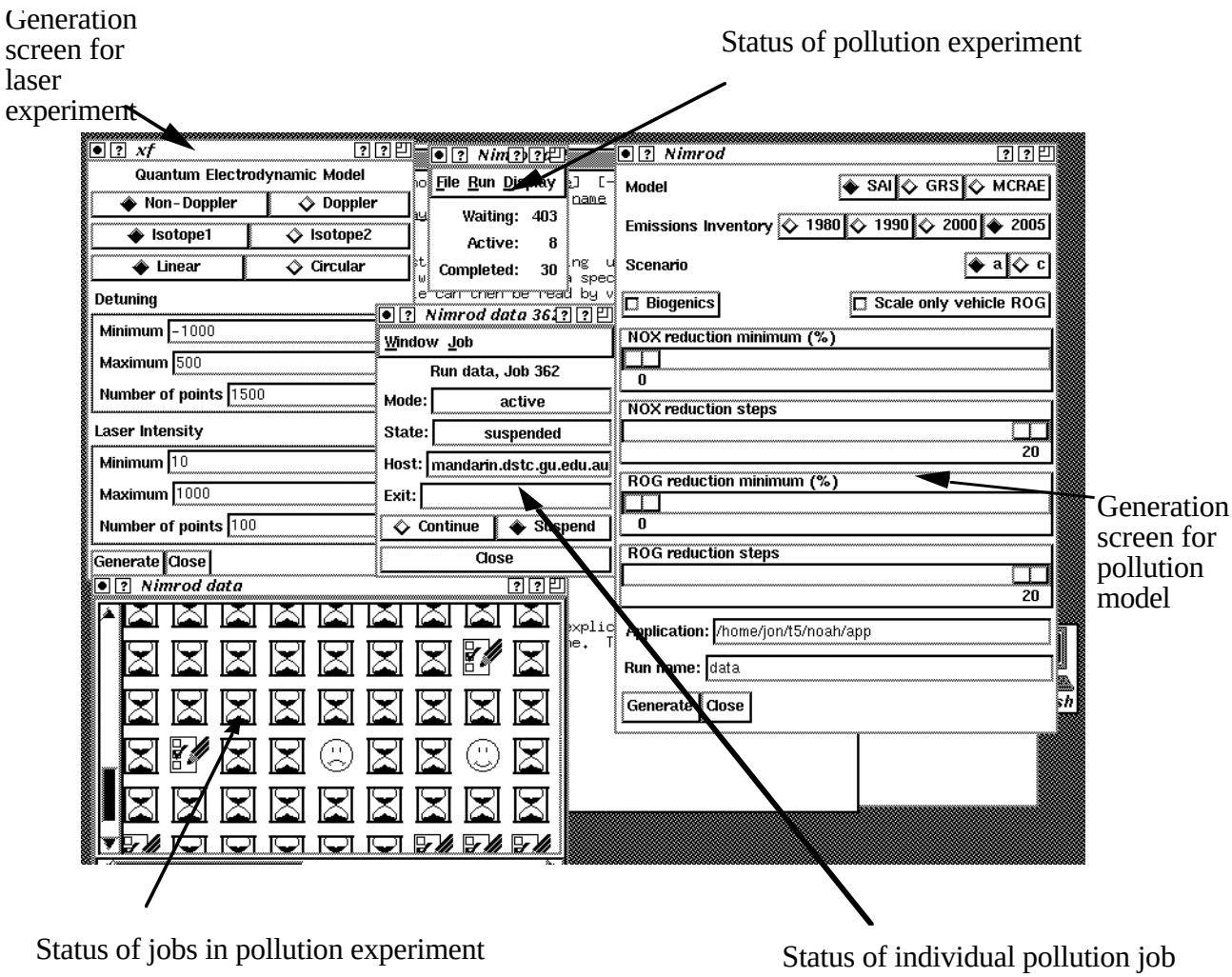


Figure 4 - Using Nimrod

1. Find a suitable remote host;

2. Perform any setup necessary to allow the job to run on the host (typically file transfer);

3. Start the job running on the remote host;

4. Wait for job termination;

5. Perform any cleanup necessary (copy output files to the originating host and remove created directories).

The task of finding suitable hosts is accomplished by connecting to a trading service [3] and specifying the requirements for a suitable host. Typical requirements are for a host of a particular architecture with a low current CPU load. The trading service then returns the names of hosts. This means that it is not necessary to know which hosts are present before Nimrod is started. Hosts may become available and unavailable during the simulation.

One of the problems of job queuing systems in general is how to specify problem-specific variables such as the files to be transferred from or to a remote host, the resource requirements of a remote host, and the execution command and environment. Most early queuing systems specified their own format for such information (Condor, NQS) and more recent entries have provided compatibility with one or more established formats, often with extensions to support additional functionality (LoadLeveler, DJM). The JDM uses Tcl [12] as a general scripting language. Tcl was chosen because it provides a thorough and clean interface to the C language, is easy extended to provide new commands, and is available on a wide variety of platforms. In this way, the user writes small Tcl scripts which specify the exact details of job setup and close down.

4.5 Job Organisation Tool

Most of the functionality of Nimrod is provided by the Job Organisation Tool. This module controls generation of a run. It provides a user interface which is related to the problem being solved, rather than to computational issues. The user selects the domains over which parameters of the problem will vary, and the system generates one job for each combination of parameters.

Once the jobs are generated and the run is started, the Job Organisation Tool allows monitoring of the progress of the run. Possible views of progress are:

1. a summary view - the main control window for the system is small enough to be left open while other work is being performed on a workstation. It provides a summary of the number of jobs waiting, active, and completed. In this way the researcher sees the current experiment which is being performed.

2. a job view - an array of icons, each icon representing one job. The shape of the icon indicates the state of the job (waiting, running, suspended, terminated normally or abnormally). More details about an individual job can be viewed by clicking on an icon. If only two parameter ranges are specified, then the position of the icon in the two dimensional display can be used to deduce the parameter values assigned to it.

3. an event view - this is a chronological list of major events that have occurred, including the launching and completion of jobs. This view is of more use to system administrators than normal Nimrod users.

The Job Organisation Tool allows the run to be directed by the user. This includes changing the priorities of jobs, suspending and continuing jobs, and terminating and restarting jobs. The Job Organisation Tool is also responsible for selection of parameter domains and controlling scripts, both of which are dependent on the problem being solved. A customised JOT is generated for controlling a particular system. We are currently developing a tool that allows flexible specification of the job parameters and automatic generation of the graphical user interface.

5 SOME EXAMPLES

In this section we illustrate the use of Nimrod for two different parametric experiments.

5.1 Photo-Chemical Pollution Modelling

This section concerns the distribution of some large scientific modelling programs which are used to compute the transport and production of photochemical smog within an urban airshed. The programs allow scientists in the Victorian Environment Protection Authority of Australia to simulate the smog production in Melbourne and to experiment with pollution reduction strategies. The same technology is currently being applied to other Australian cities.

Simulation has major advantages over direct physical experimentation. It is possible to study many more scenarios than would be physically possible, and it is also possible to measure and assess the effect of control strategies without the enormous expense of implementing them.

Photochemical smog modelling poses some interesting challenges. Such models consume enormous amounts of processor time and memory, and must often be performed within strict time constraints and budgets. Parallel and distributed computing technology has the potential to provide realistically priced platforms for performing such experiments.

One of the major uses of photochemical airshed models is to compute oxidant concentrations. Oxidants, such as ozone, are generated as a result of the chemical interaction between various precursors such as oxides of nitrogen (NOx) and other reactive organic compounds (ROCs) in the presence of ultra-violet radiation. Ozone is of particular importance because of its health related side effects; in Melbourne, Australia, the peak ozone levels have been observed to exceed 0.12 ppm in recent years, which is a widely adopted health standard level.

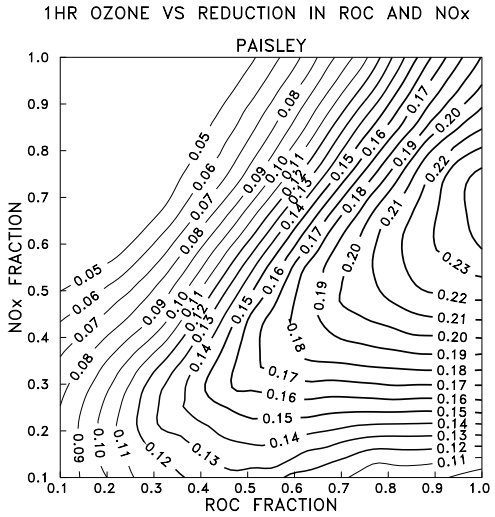


Figure 5 - Sample ozone contours

One way in which the model is used is to determine the sensitivity of the photo chemistry to various input parameters. For example, it is common to plot the peak ozone concentration as a function of NOx and ROC control, as shown in Figure 5. These diagrams then allow the modeller to determine the parameters that most affect the ozone level. The experiments are performed for a number of distinct physical locations. Unfortunately, the amount of work that is required to perform any one contour chart is enormous; if a single scenario takes about 8 hours of workstation time, then one contour chart of 50 independent runs requires 400 hours to perform. If the charts are prepared for up to 10 different weather conditions and emissions data bases, then 4000 hours of compute time are required. Thus, generation of the contour charts require access to supercomputing technology in order to produce timely results.

Rather than using conventional supercomputers to perform this work, we used a large collection of IBM RS6000 and DEC Alpha workstations, physically distributed across Brisbane (The RS6000s were part of an IBM SP2 parallel supercomputer). This utilised machines which would otherwise have been idle overnight. Thus, the work was performed without disturbing the owners of the workstations. Using this resource, it was possible to produce one ozone contour chart like the one shown in Figure 5 overnight. More details of the photo chemical pollution work can be found in [1].

5.2 Laser Atom Interaction Modelling

A detailed understanding of the collision processes between atoms, electrons and ions is of great interest in the atomic physics community. This knowledge is important in the explanation of laboratory and astrophysical plasmas, spectroscopic and surface collision physics, and scattering dynamics. Applications include fluorescent lamp and gas laser technology, surface science and atmospheric physics [2].

Of particular interest is the investigation of electron collisions with a short lived laser excited target atom. Currently there are two experimental methods for exploration of the electron-excited atom collision process. The first is the electron-photon coincidence method. A fluorescence photon from the electron excited state is detected after polarisation analysis in coincidence with the inelastically scattered electron which was responsible for excitation. The second method is the electron-superelastic scattering technique. An atom is optically prepared by a laser of known polarisation in an excited state and scattered electrons, which gain energy by collisionally de-exciting the atom, are detected.

The second method, the electron-superelastic technique, requires a detailed understanding of the laser-atom interaction as a function of laser intensity, laser polarisation and laser/atom detunings. It is possible using Quantum Electrodynamic (QED) theory, to generate equations of motion for atomic operator elements representing atomic populations in the ground and excited state, optical coherences formed between the ground and excited state by the laser and excited state coherences formed by the laser. The QED model generates closed sets of coupled, first order, linear, homogeneous differential equations. These equations are solved using numeric integration, which can be time consuming.

Once the dynamics of the atomic operators are known, it is theoretically possible to predict the line polarisation (K) for linearly polarised excitation, or the optical pumping parameter (K') for circularly polarised

excitation, as shown in Figure 6. It is how these parameters vary as a function of laser intensity and detuning that is of particular interest to physicists. Introducing integration over the Doppler profile of the atomic beam introduces another complexity which further lengthens the computing time needed.

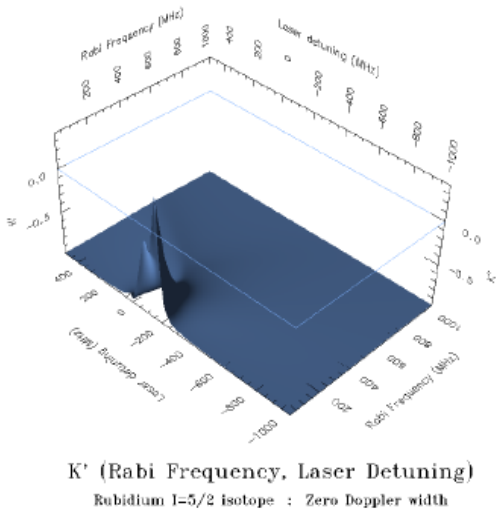


Figure 6 - Output of Laser detuning experiment

Using Nimrod, it is possible to plot K and K' as a function of laser intensity and laser detuning at the same time. A third variable, the Doppler width of the atomic beam, can be introduced and with the current software three dimensional plots can now display K and K' as functions of laser intensity, detuning and atomic beam Doppler width. This will allow the most detailed presentation of all possible data produced by these computations which covers all experimental conditions currently under investigation. As in the photo chemical pollution example, Nimrod has made it possible to distribute this application across idle machines without any consideration of parallel or distributed programming practices.

6 CONCLUSIONS

Our initial experiences with Nimrod have been very encouraging. We have successfully applied it to two different applications from different disciplines. In both cases the scientists have readily adopted the system for routine experiments. Because Nimrod shields them from the intimate details of scheduling jobs on distributed machines, and such topics as distributed file systems, they have been able to concentrate on the science underlying the experiments. In both case studies no changes were made to the application to allow it to operate in a parametric manner.

Currently, it is necessary to develop a tailored Job Organisation Tool manually using Tcl/Tk. Whilst this is not difficult for programmers with Tcl/Tk experience, it is more complex than necessary. We are currently developing a system for automatically generating JOTs using a simple specification language. This will make it possible to generate a JOT by compiling a simple description of the parameters and their ranges, together with the scripts for copying files in and out of the target system.

Whilst Nimrod is a stand alone system it is be possible to integrate its concept of automatic job generation into products such as LoadLeveler. Nimrod's dependence on DCE is limited to the software layer responsible for communications transport. This can be changed to alternative protocols with relative ease. It is worth noting that it took about one day to port Nimrod from the DEC Alpha to the IBM SP2, including the installation of DCE on the SP2. We regard this as a satisfactory outcome.

Acknowledgments

This work has been performed at Griffith University as part of the Co-operative Research Centre for Distributed Systems Technology (DSTC). The authors wish to acknowledge the support of Mr Melfyn Lloyd for proof reading a draft of this paper. Thanks go to our colleague Martin Cope, from the Victorian Environment Protection Authority and to Andrew Lewis from Griffith University. The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia

References

[1] Abramson, D.A., Cope, M. and McKenzie, R. "Modelling Photochemical Pollution using Parallel and Distributed Computing Platforms", Proceedings of PARLE-94, pp 478 - 489, Athens, Greece, July, 1994.

[2] Anderson, N, Gallagher, J.W. and Hertel, I.V., "Physics Reports (Review Section of Phys Rev Lett)", 165, Nos. 1 & 2, 1-188, 1988.

[3] Beitz, A. and Bearman, M. "A ODP Trading Service for DCE", Proceedings of the First International Workshop on Services in Distributed and Networked Environments (SDNE), June 1994, Prague, Czech Republic, IEEE Computer Society Press, pp 42-49, 1994.

[4] Butler R. and Lusk E., "Monitors, Message, and Clusters: The p4 Parallel Programming System", PARCOMP, Vol 20, pp 547-564, 1994.

[5] Bricker, A., Litzkow, M., Livny, M., "Condor Technical Summary", University of Wisconsin - Madison, October, 1991.

[6] Ferstl, F, "CODINE Technical Overview", Genias, April 1993.

[7] Green, T. P., Snyder, J. "DQS, A Distributed Queuing System", Florida State University, March 1993.

[8] International Business Machines Corporation, "IBM LoadLeveler: User's Guide", Kingston, NY, March 1993.

[9] Kaplan, J., Nelson, M. , "A Comparison of Queuing, Cluster and Distributed Comuting Systems", NASA Technical Memorandum 109025.

[10] Litzkow, M., Livny, M. and Mutka, M. Condor - A Hunter of Idle Workstations". Proceedings of the 8th International Conference on Distributed Computing Systems. San Jose, Calif., June, 1988.

[11] Litzkow, M., Livny, M., "Experiences With The Condor Distributed Batch System", Proceedings of the IEEE Workshop on Experimental Distributed Systems, Huntsville, AL, October, 1990.

[12] Ousterhout, J. K. "Tcl and the Tk Toolkit", Addison-Wesley, April, 1994.

[13] Pancake, C., Cook, C., "What Users Need in Parallel Tool Support: Survey Results and Analysis", Proceedings of 1994 Scalable High-Performance Computing Conference, May 23-25, Knoxville, Tennessee, 1994.

[14] Revor, L. S., "DQS Users Guide", Argonne National Laboratory, September, 1992.

[15] Rosenberry, W., Kenney, D., Fisher, G. "Understanding DCE.", O'Reilly & Associates, Inc. Sebastopol, California,. 1992.

[16] Shirley, J. "Guide to Writing DCE Applications", O'Reilly & Associates, Inc,. 1992.

[17] Sosic, R."Dynascope: A tool for program directing", Proceedings of SIGPLAN '92 Conference on Programming Language Design and Implementation, pages 12-21. ACM, 1992.

[18] Sun MicroSystems Inc. "SunOS 5.2 Network Interfaces Programmer's Guide", Part No 801-3927-10, May, 1993.

[19] Sunderam, V, Geist, G, Dongarra, J and Mancheck, "The PVM Concurrent Computing System: Evolution, Experiences and Trends", Journal of Parallel Computing, 20(4), pp. 531 - 546, March, 1994.

Job Management in Grids of MOSIX Clusters

David Abramson†, Amnon Barak‡ and Colin Enticott§

† School of Computer Science & Software Engineering, Monash University, Clayton, Australia, 3800
davida@csse.monash.edu.au

‡ Institute of Computer Science The Hebrew University Jerusalem Jerusalem 91904, Israel
amnon@cs.huji.ac.il

§ CRC for Enterprise Distributed Systems Technology c/o School of Computer Science & SE Monash University, Clayton Australia, 3800
Colin.Enticott@csse.monash.edu.au

Abstract

EnFuzion and MOSIX are two packages that represent different approaches to cluster management. EnFuzion is a user-level queuing system that can dispatch a predetermined number of processes to a cluster. It is a commercial version of Nimrod, a tool that supports parameter sweep applications on a variety of platforms. MOSIX, on the other hand, is operating system (kernel) level software that supports preemptive process migration for near optimal, cluster-wide resource management, virtually makings the cluster run like an SMP. Traditionally, users either use EnFuzion with a conventional cluster operating system, or MOSIX without a queue manager. This paper presents a Grid management system that combines EnFuzion with MOSIX for efficient management of processes in multiple clusters. We present a range of experiments that demonstrate the advantages of such a combination, including a real world case study that distributed a computational model of a solar system.

1 Introduction

Commodity Clusters have become the platform of choice for cheap high performance computing [9]. Typically, these are built from standard, off-the-shelf components like commodity processors and networks, and use the Linux operating system. Two software packages that have been developed for managing computations on clusters are MOSIX [1] and EnFuzion, a commercial version of Nimrod [15] by Axceleon Inc [8].

MOSIX is an add-in to Linux that supports a single system image, and makes the cluster behave almost like a symmetric multi-processor (SMP). MOSIX provides mechanisms for migrating processes between the nodes of the cluster without the process being aware of where it is executing. Importantly, it allows processes to be migrated even when they are performing input-output and interprocess communication, and contains novel heuristics for deciding where and when to migrate processes. As a result, users do not need to concern themselves with the mapping of processes to nodes, nor how to mix processes across nodes with different speeds, since these are handled transparently by MOSIX. MOSIX does not provide any high level job queuing functions, and these need to be provided by other queue management packages. Thus it cannot actually limit the amount of work that is scheduled for execution, unlike queue management systems like EnFuzion, PBS [2], LSF [9][11] and Condor [12]. Without a separate queue manager, it is possible to overload a MOSIX cluster severely.

EnFuzion, on the other hand, is a high level queue management system developed for running large parameter sweep applications on clusters. EnFuzion allows a user to specify very large computational experiments in which the same program is run multiple times, each time with a unique set of parameters. Like MOSIX, EnFuzion determines the best node on which to run a process dynamically, but then never migrates the process once it starts running. Accordingly, it is less adept than MOSIX at accounting for variability in load. Also, Neither MOSIX or EnFuzion are able to spread load across multiple clusters.

Recently there has been much interest in building large "Grids" of high performance clusters [13]. Aggregating clusters is a natural way of achieving more power than is available in any one system, without permanently combining clusters into one machine. This requirement is not uncommon in organisations in which the clusters are owned by different departments, and in which the distribution of work is not uniform. For example, if two departments each own a cluster, then it should be possible for any one department to exploit both systems at one time providing the other department does not require their cluster at that time. On the other hand, when both departments require their machines at the same time, they behave as two totally separate systems and do not share any load. However, we have developed a separate package called the Multi EnFuzion Client (MEC) [3] that distributes parameter sweep experiments to multiple machines in which EnFuzion is installed. The MEC is based on our previous work with the Nimrod/G package [14][15], and exploits many of the same approaches to scheduling across different systems. Unlike other general Grid systems, the MEC is able to operate across firewalls effectively by establishing its own virtual private network.

In this paper, we explore the combination of EnFuzion, the Multi EnFuzion Client and MOSIX, and demonstrate that there are significant advantages in combining them. When EnFuzion and MOSIX are combined on a single

cluster, the system exhibits the excellent workload management aspects of MOSIX, whilst providing a powerful user-level queue management system for parameter sweep applications. This combination is unique. When further combined with the MEC, the system can be shown to scale to enterprise wide Grids of homogeneous clusters. Importantly, the MEC works without using general Grid middleware like Globus [13]. This is only possible because we assume a common cluster architecture and operating system (namely MOSIX). Whilst not as general as Grid computing, this is still applicable in many practical circumstances.

The paper begins with a more detailed discussion of how MOSIX and EnFuzion work, followed by a description of the MEC. Then we present results that demonstrate the advantages of combining EnFuzion and MOSIX, and we show how this extends to the MEC. Finally, we demonstrate a real parameter sweep application that is executed on machines in Australia and Israel, and highlight the performance of the system.

2 Job Management in Clusters

2.1 EnFuzion

EnFuzion is an application level package that provides a high level environment for the creation, distribution and management of large parameter sweep applications. EnFuzion is a commercial code that implements research ideas explored in the Nimrod project [15]. Parameter sweep applications are characterised by the execution of many jobs, each of which explores some part of a larger parameter space. Typically, a computational model is executed many times, whilst parameters to the model are varied. Apart from an initial scattering of files and input parameters, and a final gathering of results, the jobs are independent, can be distributed to a collection of networked processors.

EnFuzion consists of two main components, a tool called the `generator` and another one called the `dispatcher`. The `generator` takes a skeleton description of a computational experiment (called a plan file), and builds a file (called a run file) that indicates how the model is to be run, and what the actual parameter combinations are. The `dispatcher` takes a run file, and schedules and runs jobs on the nodes that are available at the time. It does this by sending each job to the next available processor, using a first-come first-serve allocation strategy. Importantly, once a job starts execution on a node, it remains there until it completes. As we will see later, this simple allocation strategy can generate pathologically bad assignments when varying execution times are mixed in the one run, or when multiple users launch a number of experiments at the same time, and in which the different experiments have jobs of varying execution times.

EnFuzion does not assume that the processors share a file system, and so files are copied from the machine that launches an experiment (called the root machine) to the computational nodes. The dispatcher arranges for the files to be copied before a job starts, and then copied back after it has completed.

Whilst EnFuzion handles a number of otherwise independent jobs on a set of otherwise separate processors, it presents a view to the use of a single experiment running on a single, very high performance, computer. This attribute has made it very popular for use by scientists and engineers who wish to utilise high performance clusters, but who do not wish to write parallel programs. Accordingly, EnFuzion has been used to support a wide range of applications [16].

2.2 MOSIX

MOSIX is a software package that enhances Linux with cluster computing capabilities. The core of MOSIX includes adaptive management algorithms and a preemptive process migration mechanism that transforms the cluster into a single system parallel computing environment, almost like an SMP.

The algorithms in MOSIX support load-balancing [5], memory ushering [6], parallel I/O [7] and cluster-wide file operations [7] These algorithms monitor uneven resource usage among the nodes and if necessary, assign and reassign processes (automatically) among the nodes in order to continuously take advantage of the best available resources. The MOSIX algorithms are geared for maximal overall performance, overhead-free scalability and ease-of-use.

The granularity of the work distribution in MOSIX is the Unix process. In MOSIX, each process has a unique home-node (where it was created), which is usually the login node of the user. The system image model is a computing cluster in which every process seems to run at its home-node and all the processes of a users' session share the execution environment of the home-node. Processes that migrate to a remote (away from the home) node use local (in the remote node) resources whenever possible but continue to interact with the user's environment by forwarding environment dependent system-calls to the home-node.

MOSIX supports preemptive (completely transparent) process migration, which can migrate almost any process, any time, to any available node. After a process is migrated, a link layer at the remote node intercepts all its system-calls. If a system-call is site independent, it is executed at the remote node. Otherwise, the system-call is forwarded to the home-node, where it is executed on behalf of the process. The above scheme is particularly useful for CPU-intensive processes. The next section describes a mechanism to optimize the performance of processes with intensive I/O and/or file operations.

Unlike most network file systems, which bring the data from the file server to the client node over the network,

the MOSIX algorithms attempt to migrate the process to the node in which the file resides.

Usually, most file operations are performed by a process on a single file system. The MOSIX scheme has significant advantages over other network file systems as it allows the use of a local file system. Clearly this eliminates the communication overhead between the process and the file server (except the cost of the process migration itself).

We note that the process migration algorithms monitor and weigh the amount of I/O operations vs. the size of each process, in an attempt to further optimize the decision whether to migrate the process or not.

3 Grids of Clusters

We have built a tool called the Multi-site EnFuzion Client (MEC) that implements some of the scheduling heuristics in our previous research tool, Nimrod/G [14][15], but uses the EnFuzion dispatcher to launch jobs on individual clusters. Unlike Nimrod/G, which uses the Globus middleware layer, the MEC uses standard internet protocols like TCP/IP and tools like SSH to communicate and launch services on remote machines.

The MEC operates by launching the EnFuzion dispatcher on each cluster, as shown in Figure 1, and then balancing the jobs across them. Because many clusters are secured behind firewalls, the MEC establishes a secure connection using SSH [4]. It then sets up a number of tunnelled sockets so that all traffic passes through one port, obviating the need to punch holes in the firewall. This is in contrast to most Grid middleware layers like Globus that do not work well when firewalls are in place, which is a huge problem in Grid computing. Our solution effectively builds a virtual private network.

The MEC uses a simple heuristic which attempts to ensure that each cluster has the correct number of jobs in their queue so that all clusters to finish within a short time of each other. The algorithm measures the rate that the various dispatchers are consuming jobs, and then allocates new jobs accordingly. For example, if one cluster is consuming jobs at twice the rate of another, the former will be given more jobs to process. Job allocations are maintained by the MEC by moving waiting jobs between each of the local dispatchers' queues. This is done until all the queues are empty.

A Java applet, which is invoked from a standard browser, is available for viewing the status of the experiment at any stage during the run, and is shown in Figure 2. This is a major enhancement over EnFuzion's GUI which required that the launching node must be using X-windows and the X-windows client must remain alive through the duration of the run. The Java applet will also allow multiple people to view the run.

Using the MEC we have been able to run experiments across a number of otherwise independent EnFuzion based clusters, achieving a Computational Grid using standard commercial software.

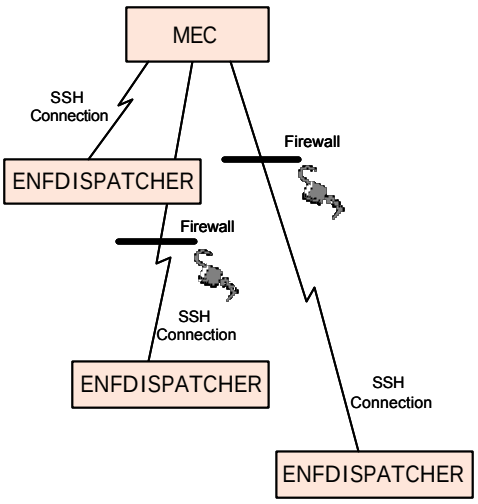
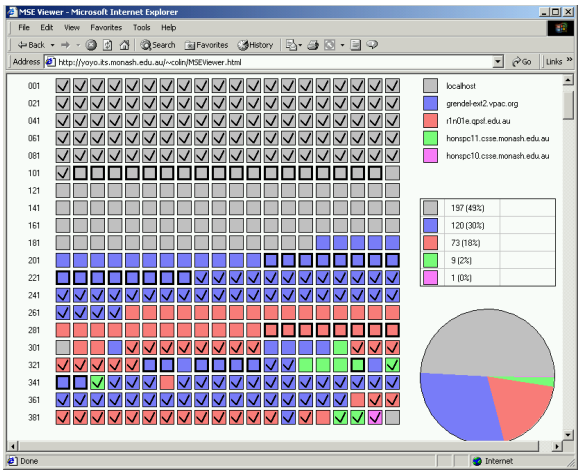


Figure 1 - Architecture of the MEC

Figure 2 - MEC progress applet



4 Combining EnFuzion with MOSIX

Combining EnFuzion and MOSIX yields a powerful platform, where EnFuzion generates , allocates and queues jobs to the cluster and MOSIX manages and optimizes the load distribution between nodes within the cluster. In particular, EnFuzion benefits from MOSIX's ability to perform preemptive process migration, and MOSIX benefits from a queue management system.

Although EnFuzion initially distributes the processes evenly across the cluster, it cannot determine how the processes will behave once started. For example, a process with one set of parameters might have very different memory requirements to ones running with a different set of parameters. In most cases, this is nondeterministic and two processes of large memory

requirements may end up on a node while another node has two processes that have little memory requirements. This may cause swapping on a node and affect the total performance of the cluster. With MOSIX's preemptive process migration, such processes could be moved to other nodes to reduce swapping. Furthermore, MOSIX will balance the cluster's load if it is not even. This will happen either while an EnFuzion run is finishing (some nodes may become idle) or the CPU speeds of the cluster's nodes are different.

Augmenting this arrangement, in which MOSIX is used to balance the load within the cluster and EnFuzion is used to queue jobs for the nodes, with the MEC, produces a system that has the ability to distribute jobs across a Grid of clusters, and then to optimize the load within each cluster. In the next section, we present the performance characteristics of such an arrangement.

5 Performance Results

This section presents the performance of several test Grid configurations, ranging from single clusters, to those which are distributed across a campus, across the metropolitan area and across continents. In each case, we executed simple tests to demonstrate the benefits of combining EnFuzion and MOSIX in the way described in the last section. The last part of the section presents tests using a real world application that simulates supernovas.

Initially, we used 2 clusters that were in the same room and were connect by a single 100Mb/sec Ethernet.

Cluster 1: four nodes, each with a dual Pentium 500MHz processor, 256MB RAM and a local disk, connected by a 100Mb/Sec hub.

Cluster 2: four nodes, each with a dual Pentium 800MHz processor, 256MB RAM and a local disk, connected by a 100Mb/Sec switch.

We executed two types of tests, one with unpredictable execution times and the other with unknown memory sizes, as follows:

CPU Test: Sixteen CPU intensive processes, with varying execution times and random arrival order. Eight processes had 1000 execution units, four processes had 200 and the remaining four had 100 execution units.

Memory Test: Eight memory intensive processes, with fixed execution times and random sizes ranging from 10MB to 200MB.

Hybrid Test: 128 CPU intensive processes, with fixed execution times and fixed size of 16MB.

None of the tests involved file I/O, unless swapping occurs in the memory test due to an allocation of processes that exceed the memory size. The results in the following tables are the completion times of all the processes, averaged over five executions.

5.1 Standalone EnFuzion and MOSIX clusters

We begin by comparing the performance of the standalone EnFuzion and MOSIX packages when each package was executed independently. In EnFuzion, the node "job limit" was set to two (which limits each CPU to one process), causing the remaining processes (if any) to queue. In MOSIX, all the processes were launched simultaneously from one node.

Table 1 shows the execution times (in Sec.) and the Standard Deviation (S.D.) of the CPU, Memory and Hybrid tests on Cluster 1 and on Cluster 2 under each of the two packages. As expected, the adaptive process management policy of MOSIX consistently outperformed the static job placement scheme of EnFuzion when the number of jobs is about the same as the number of processors. For example, the relatively long execution times of the memory test by EnFuzion are due to paging resulting from poor initial allocation of processes to nodes. Whilst it is possible to assist EnFuzion in making a better allocation, it requires some modification and tailoring of the application code and some experiments have nondeterministic memory requirements. MOSIX, on the other hand was able to correct such allocations using its "memory ushering" algorithms [6]. However, the Hybrid test highlights the behaviour of MOSIX when the number of jobs exceeds the number of processors dramatically. In this test MOSIX was not able to run the experiment because the memory requirements exceeded the available swap space. This test clearly highlights the need for a queuing system with MOSIX. We did not bother running this test on both clusters.

Table 1. Standalone EnFuzion and MOSIX times

Test	Package	Cluster 1		Cluster 2	
		Time	S.D.	Time	S.D.
CPU	MOSIX	921	1	580	1
CPU	EnFuzion	1300	121	970	87
MEMORY	MOSIX	316	58	303	133
MEMORY	EnFuzion	2976	3987	3568	3561
Hybrid	MOSIX	Would	not run	Not done	
Hybrid	EnFuzion	784	3	Not done	

5.2 Execution on a single cluster

As shown in 5.1 EnFuzion has a queuing capability that controls the number of processes that are dispatched to the nodes of a cluster. MOSIX, on the other hand is better suited to manage already allocated processes throughout the execution. By combining the two packages, one can use EnFuzion to dispatch a predetermined number of processes to a cluster, then to use MOSIX to manage the processes within a cluster, to best utilise the resources throughout the execution.

To check this idea in a single cluster, we used EnFuzion to launch the processes and compare the results with MOSIX enabled or disabled. For the CPU test, the EnFuzion "job limit" was set to four (two processes per

CPU) while the memory test was executed with a job limit of two (one process per CPU, as in the previous test).

The results, presented in Table 2, show that the execution times of EnFuzion with MOSIX enabled are 26-2500% better than the times with MOSIX disabled. We note that the large standard deviation (S.D.) of the EnFuzion with MOSIX disabled is due to EnFuzion allocating the initial jobs based on the order and timing of the nodes launching (effectively random).

Table 2. Execution times (Sec.)

Test	EnFuzion with MOSIX	Cluster 1		Cluster 2	
		Time	S.D.	Time	S.D.
CPU	Enabled	917	1	578	2
CPU	Disabled	1157	84	958	235
MEMORY	Enabled	290	68	205	67
MEMORY	Disabled	3260	67	5203	3516

Without MOSIX, we can see that the EnFuzion times for the same CPU tests with "job limit" of four (CPU test with MOSIX disabled in Table 2) were up to 12% faster than with a "job limit" of two (CPU test with EnFuzion in Table 1). The CPU tests indicate that a shorter completion time could be obtained by running multiple processes per processor. This will be discussed in 5.3.

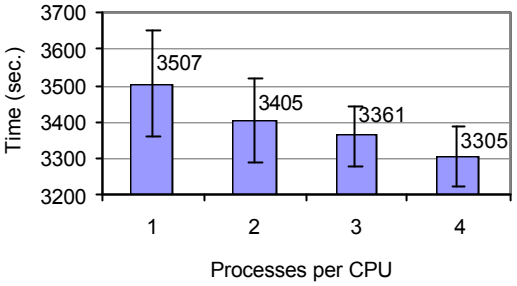
In the remaining of this section, we executed only the CPU test. All the experiments were conducted by using the MEC with one EnFuzion dispatcher per cluster and with MOSIX enabled in each cluster.

5.3 How many processes per CPU?

Comparing the times of the CPU test of EnFuzion (without MOSIX) in Table 1 and Table 2 shows an improvement by increasing the number of processes per CPU. Increasing this further without MOSIX could cause nodes that have accumulated longer processes to finish later and thus to a longer completion time. The MOSIX's ability to migrate processes removes this problem and thus the possibility to overload the cluster.

In the next test, we increased the total number of processes to 128, adjusted the number of processes per CPU in each benchmark from 1 to 4 and executed on Cluster 2. Figure 3 shows the performance gained by increasing the load of the cluster. From the figure it can be seen that overloading each node with more than one process actually resulted in increasingly better performance. Obviously, this effect does not continue beyond a certain small number of processes. One explanation for this phenomenon is that having a greater number of processes executing ensures that the longer processes had shorter remaining execution by the time the load reduces to 1.

Figure 3. Varying load on a MOSIX cluster

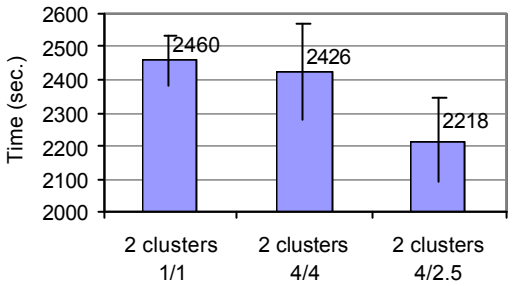


5.4 A 2-cluster Grid

In this experiment, we executed the CPU tests on Cluster 1 and Cluster 2 (totalling 8 nodes) forming a "same floor", 2-cluster Grid. Each cluster was managed by its own MOSIX and the Grid allocation of processes was done by the MEC and the EnFuzion dispatcher. We executed this experiment with 1 process per CPU, 4 processes per CPU and with a weighted combination of 2.5 processes per CPU on cluster 1 (5 processes per node) and 4 on cluster 2.

Figure 4 shows a slight improvement moving from 1 process per CPU to 4 processes per CPU. This was expected from the results of the previous section. During these tests, it was observed that the faster cluster finished long before the slower one. Consequently, we implemented a more sophisticated load balancing algorithm, in which the load was allocated relative to the CPU power of the clusters, allowing them to complete at approximately the same time. As a result of this, the jobs completed about 9% earlier.

Figure 4. Different loads on two MOSIX clusters



Comparing Figure 3 with Figure 4, shows the speed increase by adding the 500MHz cluster to the 800MHz cluster in a small Grid. This combinations of machines has a lower bound for the execution time of 2033 seconds, however, due to the granularity of the processes execution time, this cannot be achieved. Using a balanced load gave the closest result.

5.5 A 5-cluster grid

At this stage, we introduce three new clusters:

Cluster 3: Four dual CPU 700MHz nodes connected by 1000Mb/sec. switch. This cluster was located on another Monash campus, 10km away.

Cluster 4: Eight single CPU nodes of varying speeds, (four 1GHz, three 750MHz and a 500MHz node). They were all connected by a 100Mb/sec. switch. These machines were located at the Victorian Partnership for Advanced Computing (VPAC), across town 20km away.

Cluster 5: Eight single CPU nodes, all with 1GHz CPUs and connected by 100Mb/sec. switch. This cluster was located at the Hebrew University of Jerusalem, over 13,500 km away - on the other side of the world.

Figure 5. Increasing the number of clusters

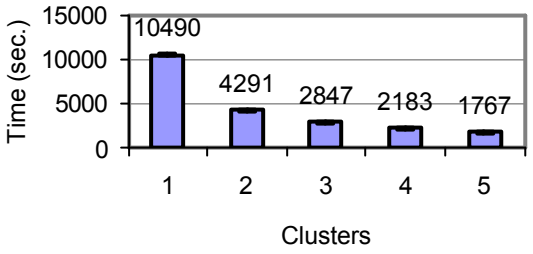


Figure 5 shows the gain achieved by adding clusters to help process the jobs. Increasing the number of CPUs to work on the problem reduces the total time, in spite of the communication delays between the clusters. By balancing the number of executing processes on each of the cluster, each cluster would finish processing at a similar time, producing the quickest complete results. These results average 338 seconds over the theoretical best. This value is also half the average duration the execution time process on the slowest cluster proving that the increase of time is caused by the granularity of processes.

5.6 A real-world test

The tests performed so far are designed to explore theoretical boundaries of cluster and Grid computing. In this section we tried a real world application. This experiment uses a code developed by Dr Kurt Liffman from the Centre for Stellar and Planetary Astrophysics (CSPA) at Monash University [17]. "The code models the early solar system, where the protoSun is surrounded by an accretion disc, which we call the solar nebula. It is thought that the planets formed from the solar nebula approximately 4.56 billion years ago. From the magnetic interaction between the Sun and the surrounding disc, a high speed, bipolar jet flow (speed ~ 200 km/s) is formed from the inner regions of the disc. Such jet flows are ubiquitous amongst early forming stellar system. The typical structure of the jet flow is that two jets are formed in the inner region of the disc, where the jet flows are pointing in a direction perpendicular to the disc. About 500 small silicate and metal spheres (size ~ 0.1 mm) are injected into the jet flow and their subsequent motion is monitored.

Each experiment executes this program 1100 times exploring over half a million particles. Typically, such particles are ejected by the jet flow and move across the accretion disc. Some particles re-enter the accretion disc, while others are ejected from the system. The purpose of the code is to measure the abundance of iron to silicon in the ejected and recaptured particles"[1]. This type of computation is ideal for EnFuzion, because each computation is uncoupled and the results can be aggregated at the end.

Table 3. Resources available for real world experiment

Cluster	Location	Dist. (km)	Number of Processors	Total GHz
Monash 1	Monash Caulfield	0	16	10.4
Monash 2	Monash Clayton	10	30	21.8
VPAC	Melbourne	20	7	6.3
Hebrew Univ	Jerusalem	13,500	48	40.0

For the execution of this experiment, we used all available resources, including 23 dual processor nodes at Monash University, 7 single processor nodes at VPAC and 39 dual and single processor nodes at Hebrew University. See Table 3 for more details.

Table 4. Results from real world application

Total GHz	Resources	Execution time (hrs)	Theoretical best (hrs)	Efficiency
1	One computer	520	520	100%
38.5	All but Hebrew Univ	14.2	13.5	95%
78.5	All	6.75	6.6	98%

Executing all 1100 processes on a single 1 GHz node took about 520 hours to complete. The resources shown in Table 3 produced the results in Table 4. The theoretical best time are calculated from dividing the time to execute on a 1GHz processor by the total computational power, as measured in total GHz. We are able to do this because the application actually scales well with the speed of the processor. From the results in Table 4, we were able to achieve close to the approximated best time. Once again, this was within the time of the average process duration on the slowest node (56 minutes).

[1] This text was supplied by Dr Liffman.

The results illustrate that the MEC was effective in distributing the load across a number of independent clusters whilst taking advantage of the MOSIX local cluster management.

6 Conclusions

In this paper we have explored the combination of two otherwise separate packages that present a view of a multi process cluster as a single computational resource. The results of a number of controlled experiments highlight scenarios in which each package is superior. Specifically, MOSIX performs better workload allocation than EnFuzion when the number of jobs is comparable with the number of processors. This is because MOSIX can reassign the processes dynamically. On the other hand, EnFuzion's queueing capability performs better when the number of jobs is much larger than the number of processors because it avoids overloading the processors. In the limit, MOSIX is unable to run experiments that require too much of a single resource, like swap space. When combined, we achieve the best of both systems.

By using the Multi EnFuzion Client we are able to combine clusters into a computational Grid. Using this we reduce the execution time of a large computational experiment that would have taken over 500 hours, to one that ran in under 7 hours, using resources distributed across the globe, and running at an efficiency of 98%. This demonstrates the combination is practical and useful for real computational science experiments.

The work in this paper has highlighted the potential for queue managers to benefit from information that MOSIX normally keeps in the execution of a process. For example, we are interested in exploring the idea of exporting dynamic parameters such as memory usage, processor utilisation, etc, to improve the allocation of jobs to clusters in the first instance.

Further, EnFuzion, like other queue management systems, allocates processes to processors so that the node is not overloaded. However, in section 5.3 we demonstrated that this under states the amount of work a machine can do, and that sometimes it is possible to place more processes on the machine. In our work, we were able to determine the optimal number of processes per processor through experiment, and then we instructed the MEC on how to allocate processes. In the future, this should be done automatically. Accordingly, we plan to produce a queue scheduler that works more closely with MOSIX in allocating.

Another area of interest is in using MOSIX's process migration techniques for deciding when, and how, a process can be migrated from one cluster to another one. This is a complex topic and requires further research.

In addition, we reported a new software tool called the Multi EnFuzion Client (MEC), which draws on research results from the Nimrod project. The MEC supports the aggregation of multiple homogeneous clusters into a Grid. Unlike Nimrod, the MEC works in a "standard" Internet environment without special Grid middleware like Globus, and without special security requirements. This is possible only because we assume a homogeneous machine base, whereas general Grids are mixes of heterogenous machines, operating systems and environemts. Further, the MEC is able to operate in the presence of firewalls unlike much other grid middleware.

Acknowledgment

We thank Glen Pringle and David Arnold and Arkady Zaslavsky at Monash University for their support, David Bannon and Sudarshan Ramachandran at the Victorian Partnership for Advanced Computing, Danny Braniss, Assaf Spanier, Ephraim Silverberg and Chana Slutzkin at Hebrew University, Kurt Liffman from the C.S.I.R.O. This project is supported by the CRC for Enterprise Distributed Systems and GrangeNet.

References

[1] http://www.MOSIX.org

[2] http://www.openPBS.org

[3] Multi EnFuzion Client, http://www.csse.monash.edu.au/~cme/mec

[4] SSH, http://www.ssh.com

[5] Barak, A., La'adan, O. and Shiloh, A., "Scalable cluster computing with MOSIX for Linux", Proc. 5-th Annual Linux Expo, Raleigh, NC, pp. 95-100, May 1999.

[6] Barak and Braverman, "Memory ushering in a scalable computing cluster", Journal of Microprocessors and Microsystems, 22 (3-4), pp. 175-182, 1998.

[7] Amar, L., Barak, A. and Shiloh, A., "The MOSIX parallel I/O system for scalable I/O performance", Proc. of the 14th International Conference Parallel and Distributed Computing and Systems (PDCS 2002), Cambridge, MA, pp. 495-500, November 2002.

[8] http://www.axceleon.com.

[9] Sterling, T., "Beowulf cluster computing with Linux", MIT Press Cambridge, MA, 2001, ISBN:0-262-69274-0

[10] http://www.platform.com

[11] Zhou, S., Zheng, X., Wang, J. and Delisle, P., "Utopia: A load sharing facility for large, heterogeneous distributed computer systems", Software-Practice and Experience, 23(12), pp 1305-1336, December 1993.

[12] Livny, M., High-Throughput Resource Management. In The Grid: Blueprint for a Future Computing Infrastructure. Morgan Kaufmann Publishers, 1999.

[13] Foster, I. and Kesselman, C., Globus: A Toolkit-Based Grid Architecture. In The Grid: Blueprint for a Future Computing Infrastructure. Morgan Kaufmann Publishers, 1999.

[14] Abramson, D., Buuya, R. and Giddy, J., "A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker", to appear, Future Generation Computer Systems. Volume 18, Issue 8, Oct-2002.

[15] Abramson, D., Giddy, J. and Kotler, L., "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", International Parallel and Distributed Processing Symposium (IPDPS), pp 520-528, Cancun, Mexico, May 2000.

[16] Abramson, D., "Parametric Modelling: Killer Apps for Linux Clusters", The Linux Journal, #73, pp 84 – 91, May 2000.

[17] Centre for Stellar and Planetary Astrophysics (CSPA), http://www.maths.monash.edu.au/astro/

Feature

Parametric Modelling: Killer Apps for Linux Clusters

Get ready for parallel processing with the Monash University.

by David Abramson

After nearly 20 years of research and development, there still has not been a wide-scale uptake of parallel computing technology. However, with recent advances in PC-based hardware and networking products, it is now possible to build a parallel computer from industry-standard, commercial, off-the-shelf products. Approaches such as those used in the Beowulf project (see http://beowulf.gsfc.nasa.gov/) advocate the connection of a number of Linux-based workstations, or high-end PCs, with high-speed networking hardware and appropriate software as the basis for a viable parallel computer. These systems support parallel programming using a message-passing methodology. In spite of this, there is still a dearth of good, widespread, parallel applications.

I believe there are two main reasons for the lack of applications. First, rapid changes in hardware architecture and a lack of software standards have made it difficult to develop robust, portable and efficient software. Second, much of the effort in developing a parallel program has been focused on low-level programming issues, such as how to support message passing in a portable manner. Consequently, insufficient attention has been given to high-level parallel programming environments built around niche application domains. Thus, application developers building parallel programs have been forced to consider very low-level issues, which are far removed from their base disciplines such as physics, chemistry and commerce.

In 1994, some of my colleagues and I began a research project called Nimrod, with the goal of producing a parallel problem-solving environment for a particular niche class of application, namely parametric modelling experiments. We were motivated to do the work for this project after experiencing the frustration of trying to perform some large-scale computational-modelling experiments on a distributed computer using the available tools. We were either forced to perform the work manually, or use low-level parallel programming tools like PVM (Parallel Virtual Machine) and batch queueing systems. Neither of these automated methods matched the type of work we wanted to perform, which at the time was modelling air pollution and exploring different control strategies. The project led to the development of a commercial tool called EnFuzion (see http://www.turbolinux.com/), which runs on a variety of UNIX platforms including Linux. It has also led to a number of very successful applications, with demonstrated returns to the researchers involved. Other types of problems can be formulated as parametric experiments, and thus can also take advantage of EnFuzion.

Parametric Modelling

Parametric modelling is concerned with computing a function value with different input parameter values. The results make it possible to explore different parameters and design options. The broad approach has been made very popular by the use of spreadsheet software, in which many different options can be rapidly evaluated and the results displayed. However, rather than simply computing a function value, we are interested in running an executable program, and thus the time required to explore a number of design scenarios may be extensive. For example, a financial model may take on the order of one hour to compute one scenario on a PC. Accordingly, to consider 100 different scenarios requires 100 hours, or over four days of computing time. This type of experiment can be feasible only if a number of computers are used to run different invocations of the program in parallel. For example, 20 machines could solve the same problem in about five hours of elapsed time.

In spite of the obvious advantage to using multiple machines to solve one large parametric experiment, almost no high-level tools are available for supporting multiple executions of the same program over distributed computers. Spreadsheet software is not designed for executing programs concurrently or utilizing distributed computers. It is possible to write a program using low-level parallel programming tools like PVM (see http://www.epm.ornl.gov/pvm/), Message Passing Interface (see http://www.mpi-forum.org/) and Remote Procedure Call (RPC), to support its execution across many machines; however, this approach has a number of disadvantages.

- First, the source must be available for the program, and it must be modified to support parallel execution. This is not always possible or desirable for complex commercial software.
- Second, all aspects of distributing the task must be built into the application. Application programmers are usually not expert in both their own discipline and parallel programming; in short, it is difficult.

- Third, unless fault tolerance is built into the application, the resulting program will not be able to handle failure in the network or in individual machines. Such fault tolerance significantly complicates the application.

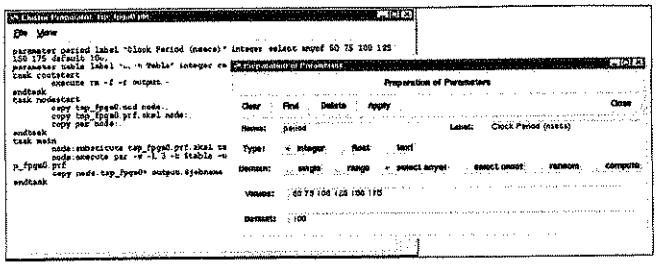


Figure 1. Plan File

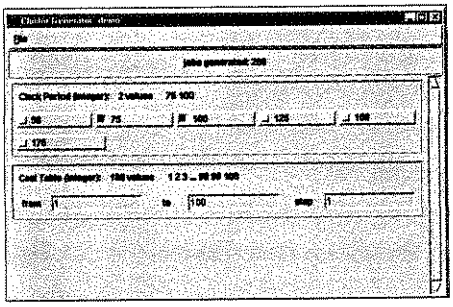


Figure 2. Generator/CAD Interaction

Accordingly, it is not surprising that few instances of this approach are used in practice.

The alternative method is to use a remote job distribution system like Platform Computing Load Sharing Facility (see http://www.platform.com), Network Queueing System (see http://www.shef.ac.uk/uni/projects/nqs/) or Portable Batch Scheduler (see http://pbs.mrj.com/). These systems allow a user to submit jobs to a virtual computer built from many individual machines. While more general than the previous parallel-programming approach, the main disadvantage of remote queueing systems is they are targeted at running jobs, not necessarily performing a parametric study. Thus, you have to build additional software for running generated jobs based on the different parameter values and after aggregating the results. This can expose the user to a number of low-level system issues, such as the management of network queues, location and nature of the underlying machines, availability of shared file systems, the method for transferring data from one machine to another, etc.

Enter TurboLinux EnFuzion

EnFuzion belongs somewhere between a user-level tool and a software development environment. EnFuzion has been designed to make it possible to build a fault-tolerant, distributed application for a parametric-modelling experiment in a time on the order of minutes. In many cases, EnFuzion requires no programming; all you do is describe the parameters to the system and give some instruction on how to run the programs. EnFuzion manages the instantiation of your code with different parameter values, sends the files to the remote machines and retrieves them, and finally, it distributes the execution. If a program or system fails, EnFuzion automatically reruns the program on another machine.

Running an EnFuzion experiment requires three phases: preparation, generation and distribution. During preparation, you develop a descriptive file called a plan. A plan contains a description of the parameters, their types and possible values. It also contains commands for sending files to remote machines, retrieving them and running the job. EnFuzion provides a tool called the Preparator which has a wizard for generating standard plan files, as shown in Figure 1. Alternatively, if you are prepared to learn EnFuzion's simple scripting language, it is possible to build a plan using a normal text editor. The plan shown in Figure 1 is typical of a simple experiment and is quite small.

The plan file is processed by a tool called the generator, which asks the user to specify the actual values for the parameters. For example, a plan file might specify that a parameter is a range of integers, without giving the start, end or increment values. The generator tool allows the user to fill in these values. It then reports how many independent program executions are required to perform the experiment by taking the cross product of all parameter values. Figure 2 shows a sample generator interaction with a CAD tool, where the clock period parameter is set to values of 75 and 100 nanoseconds and the cost table parameter is varied from 1 to 100. This interaction

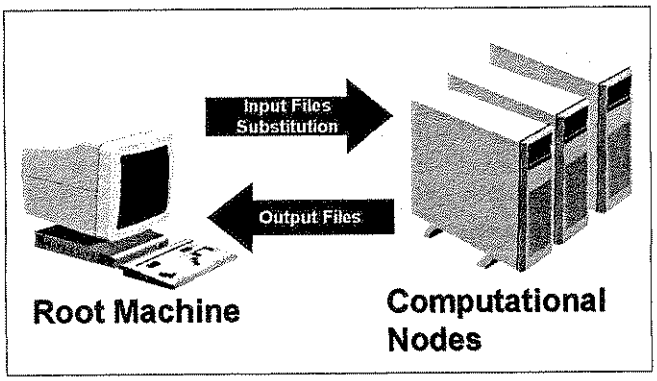


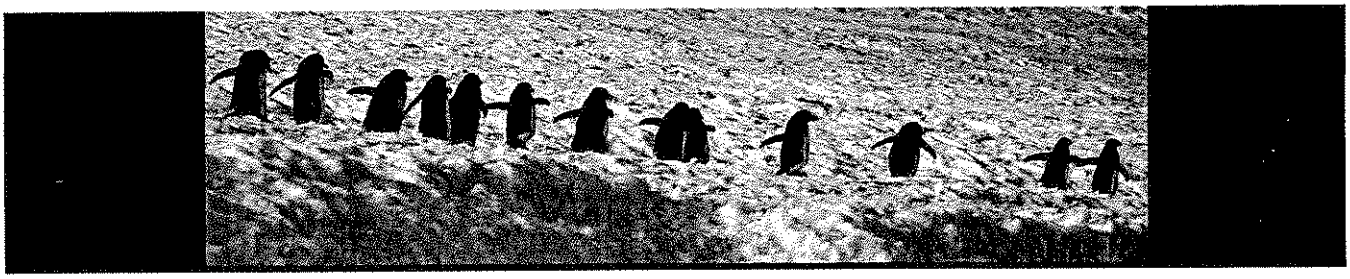
Figure 3. Computational Nodes



Figure 4. Monash Parametric Modelling Engine

Table 1. Applications of the MPME

Problem	Researchers
Evaluation of a compression algorithm based on byte-pair encoding	School of Computer Science and Software Engineering, Monash University
Simulation of financial operation of a small manufacturing company	School of Business Systems, Monash University
Simulation of fuzzy-logic controller for ATM networks	School of Computer Science and Software Engineering, Monash University
Mobile-phone-frequency assignment using genetic algorithms	School of Business Systems, Monash University
A genetic algorithm for solving the N Queens problem	School of Business Systems, Monash University
Modelling computer networks using OPNET	School of Computer Science and Software Engineering, Monash University
Hopfield Neural Networks for solving Timetabling problems	School of Business Systems, Monash University
Fracture Simulation in 2-dimensional models	Department of Mechanical Engineering, Monash University
Simulation of large Ad Hoc Mobile Networks	School of Computer Science and Software Engineering, Monash University
Monte Carlo simulation of ionizing radiation equipment	Australian Radiation Protection and Nuclear Safety Agency
Public health policy simulation for injecting drug management	School of Business Systems, Monash University and MacFarlane Burnett Centre for Medical Research
Shake simulation of F111 airframes	School of Computer Science and Software Engineering, Monash University and Australian Department of Defense



Searching for A Complete Fax Server for Linux? Just Follow The Leader...

For over a decade, VSI-FAX has been widely recognized as the undisputed leader in Unix-based fax software, and now it's ready for Linux. VSI-FAX 4.0 for Linux provides a complete solution for both small departments AND large enterprise environments. Use VSI-FAX's powerful development tools (Software Development Kit and integration mechanisms) to automate faxing from any application. You can fax merge from StarOffice, WordPerfect Office 2000, Lotus SmartSuite and Microsoft Office 2000. Plus, VSI-FAX is certified for use with Red Hat Linux and Caldera OpenLinux. So call us or visit us on the web today to get your free evaluation copy, and see for yourself why VSI-FAX for Linux is by far the easiest way to fax from Linux.

VSI-FAX™

Download your FREE Evaluation Copy Today!

www.vsi.com/fax4linux
800.556.4874

vsi™
Bridging the Gap between Business and E-Business™

© Copyright 2000 V-Systems, Inc. The VSI logo, VSI-FAX logo, and "Bridging the Gap Between Business and E-Business" are trademarks of V-Systems, Inc.

generated 200 individual executions of the CAD package.

The generator produces a run file, which contains all the information regarding what parameter values are to be used and how to run the jobs. This file is processed by a tool called the dispatcher, which organizes the actual execution. EnFuzion calls the machine on which you develop your plan the root machine. The work is performed on a number of computational nodes, as shown in Figure 3. Files are sent from the root machine to the computational nodes as required. Output is returned to the root for postprocessing.

The dispatcher chooses to send work to machines which are named in a user-supplied file, so every user can have a different list of machines. The dispatcher contacts the nodes and determines whether it is possible to start execution of the tasks. EnFuzion allows you to restrict the number of tasks run by using many different thresholds, such as a maximum number of tasks, the hours a node will be available and the peak load average for the machine. At Monash, we have augmented EnFuzion with a simple scheme using the UNIX command **nice**. This allows a node to run more tasks than available processors, but long-running jobs are "niced" to allow short ones to have a higher priority than long-running ones. This seems to be a good way of mixing short- and long-running jobs without restricting the job mix artificially.

The Monash Parametric Modelling Engine

At Monash University, we have built a cluster of 30 dual-processor Pentium machines running Linux, called the Monash Parametric Modelling Engine (MPME). A single machine acts as the host, and is connected to both the public network and a private 100Mbit network for linking the computational nodes. Figure 4 shows part of this machine.

Each MPME node runs a full Linux Red Hat 5.2 kernel, and the standard GNU tools, such as gcc, gdb, etc. Typically, users log in to the host and use EnFuzion to schedule work on the nodes. We have configured the system to accept up to five jobs per node, even though each node contains only two processors. To control the load on each machine, a script is run which increases the nice level of each process the longer it executes. This means it is possible to mix long- and short-running jobs on the platform. In contrast, when we limited the number of jobs to the number of processors, we found that long-running jobs were monopolizing the nodes and short-running jobs were rarely run.

To date, we have used the MPME to support a wide range of applications. Table 1 shows a list of the applications mounted during 1998 and 1999. Many of these are student projects completed as part of a semester subject. In the sidebar "A Case Study", one of our postgraduate students, Carlo Kopp, describes use of the cluster to perform his network simulations. The results have been quite dramatic, in this case the additional computational resources allowed him to explore many more design options than he initially thought would be possible.



David Abramson (davida@dgs.monash.edu.au) is the head of the School of Computer Science and Software Engineering at Monash University in Australia. Professor Abramson has been involved in computer architecture and high performance computing research since 1979 and is currently project leader in the Co-operative Research Centre for Distributed Systems Nimrod Project. His interests include high-performance computer systems design, software engineering tools for programming parallel and distributed supercomputers and stained glass windows.

Resources

EnFuzion is described on the TurboLinux web page at http://www.turbolinux.com/.

The Monash Parametric Modeling Engine is described at http://hathor.cs.monash.edu.au/.

The early work on Nimrod and current work on a new global version of Nimrod called Nimrod/G can be found at http://www.csse.monash.edu.au/~davida/nimrod.html.

See **A Case Study** on page 90.

Enhanced Security (SSL), Speed & Reliability

Paymentplus

The Leader In Global Payment Solutions

- FIRST-TIER UNIX SUPPORT
- Real-Time Authorizations (Under 6 Seconds)
- Host and User Authentication
- All Major Credit Cards
- Electronic Checks

Visit our website: www.paymentplus.com

Contact us:
sales@paymentplus.com
tel: (877) 4-PAYPLUS

Reston, VA • Salem, NH
New Delhi, India • Pune, India
Melbourne, Australia

A Case Study: Parametric Modelling of Propagation Effects in Large Ad Hoc Networks

by Carlo Kopp

The largest and most complex parametric simulation performed on the Monash CSSE MPME cluster was done as part of a research project dealing with microwave propagation performance effects on large ad hoc microwave networks.

Ad hoc networks are a category of mobile network in which no fixed infrastructure is used; every mobile system in the network acts as a router on behalf of its peers. The topology of such networks is dynamic; it changes continuously as vehicles within the network drift in and out of range of one another. It also changes as a result of weather impairing the propagation of microwave links between these vehicles.

Simulating this behaviour is very demanding computationally. Finding a link between two stations requires the solution of a nonlinear differential equation to account for atmospheric refraction. This is performed using a multistep polynomial method to solve an initial value problem. The shooting method is then employed iteratively to find the solution of the refractive equation. Once we know the shape of the curve, we must perform an atmospheric loss calculation which requires computation of losses due to oxygen resonance, water-vapour resonance, and scattering in cloud and rain, all performed for very small increments of beam altitude above the surface of the earth.

The complete simulation entails loading a database of geographical positions of vehicles in the network from disk, a search through this database in which every vehicle is tested against every other to discover the existence of links, and the creation of a link database. This operation is essentially proportional to N^2, where N is the number of vehicles in the network.

Once we have a database of links, we use this population to solve an all-paths/shortest-path problem to find the shortest route connecting every pair of vehicles in the ad hoc network. This compute load is proportional to N^3 and is also time-consuming. We may then extract these routes and look at properties such as the number of hops between specific vehicles or specific ground stations in the network, and aggregate behaviour such as the net availability of routes over time as the topology of the network varies.

Needless to say, this is a very demanding simulation. On a 333MHz Pentium II CPU, it takes hours to complete for a population of hundreds of nodes, and several days to complete for populations of thousands of nodes.

The general problem is one of exploring the behaviour of the network, using different frequencies for the microwave links, different weather conditions and different possible link speeds. Because the system we are trying to model is so complex, only a comprehensive simulation across a large range of input parameters yields a trustworthy solution.

In the early phases of this project, I envisaged doing this on a single top-end Pentium II processor. When confronted with the sheer volume of computation, I had to craft a simplified version of the intended simulation which, in effect, cheated and used a very simple model to handle the propagation behaviour. Professor David Abramson proposed that I employ the cluster instead—I was initially very sceptical and resisted. However, after much argument, David prevailed, and I ported the simulation from FreeBSD to Linux and developed the EnFuzion plan scripts to run it.

It was quickly apparent that having 20 CPUs allowed me to grind through the volume of computations 20 times faster. Once I gained confidence with the cluster, I incorporated the accurate propagation model and expanded the scope of the simulation to map out the whole range of frequencies, link speeds and weather models ideally required to solve the complete problem. At the time of this writing, the project is virtually complete, and about 950 simulation runs have been performed.

Use of the cluster allowed me to demolish the performance bottleneck which had plagued me in the traditional world of single-processor simulations. The simple scripting language used by EnFuzion allowed me to organize the parameter space and efficiently conduct the simulations without having to sit over the machine, waiting for one job to complete in order to fire off the next one.

To say we did not have teething problems would be untrue. The simulation had to be thoroughly debugged, and some adaptations were required to turn off the X11-based screen display features. The biggest problems we faced initially were managing the load on the system, and tweaking the Red Hat Linux installations to achieve the required reliability.

Solving the load management problem required some brainstorming. Executing long-running jobs of this complexity, across all 20, later 28 and finally 60 CPUs in the cluster, tended to discourage casual daytime users who were confronted with a load-saturated system. After being spammed by a mob of irate users, I proposed to David that a scheduler was required. David suggested that the UNIX renicing mechanism might be usable; after some debate on implementation, our system administrator crafted a **cron** job to do this. The irate users were pacified, and the simulations continued successfully.

The biggest source of difficulty was the lack of robustness in the Red Hat networking protocol stack running on the root node. This was eventually solved. For very large clusters, the root server must be rock-solid reliable.

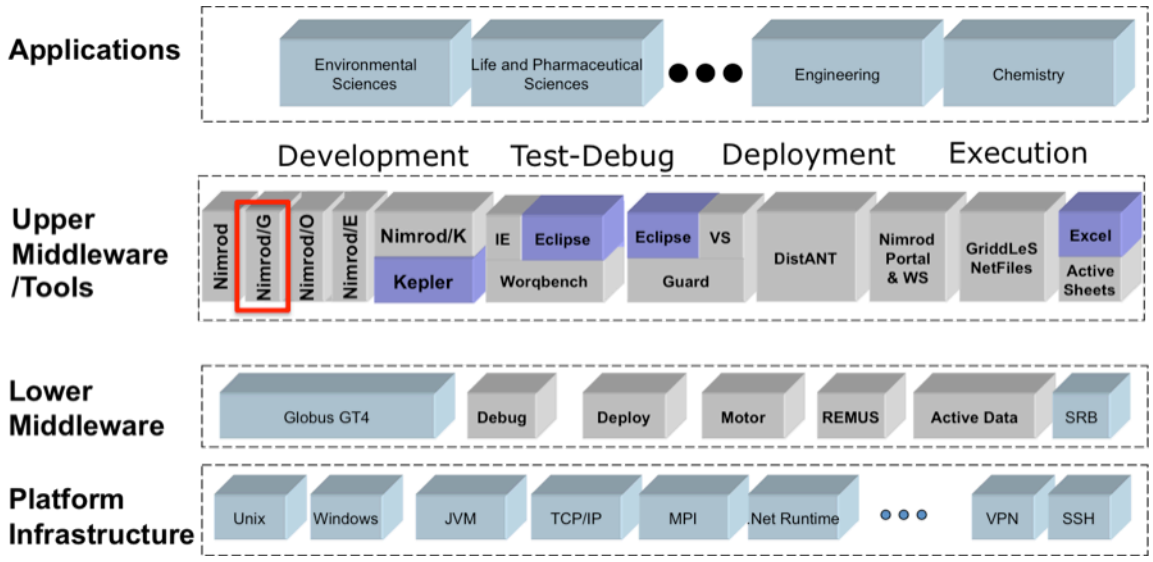
In summary, the use of the cluster converted a would-be simulation nightmare into a relatively straightforward exercise. Because the simulation did not lend itself to vectorization, a Cray Y-MP was not a viable solution to this dilemma, budgetary issues aside. In the end, the problem was solved by using a pair of 19-inch racks loaded with commodity Pentiums running freeware Red Hat Linux.

After my initial scepticism, it was a case of eating humble pie—the boss did know better, after all! Clustering works very effectively, and for problems which don't vectorize easily, it may be the only viable solution.



A native of Perth, Western Australia, Carlo Kopp graduated with first class honors in Electrical Engineering in 1984, from the University of Western Australia. In 1996 he completed an MSc in Computer Science and has submitted a PhD in the same discipline, at Monash University. He has consulted in UNIX systems programming, performance engineering and system administration. He now lectures computer architecture at Monash University.

Chapter 3: Nimrod/G and the Grid Economy



In 2000 I extended the resource base for parametric computing to encompass globally distributed machines. Apart from engineering the distributed computing infrastructure itself, this work proposed and developed the idea of using a computational economy to underpin the scheduler. Paper (7) discusses the original infrastructure changes required to achieve this and paper (8) proposed the idea of a computational economy. This original idea was developed further by my PhD student, Buyya, and is reported in papers (9) and (10). In paper (11) we enhanced the scheduling algorithms to support fault tolerance, and this was performed with my PhD student, Ayyub.

(1) Abramson, D and Giddy, J. "Scheduling Large Parametric Modelling Experiments on a Distributed Meta-computer", PCW '97, September 25 and 26, 1997, Australian National University, Canberra, pp P2-H-1 – P2-H-8 — 3-1 .. 3-8

(2) Abramson, D., Giddy, J. and Kotler, L. "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", International Parallel and Distributed Processing Symposium (IPDPS), pp 520- 528, Cancun, Mexico, May 2000 — 3-9 .. 3-17

(3) Buyya, R., Abramson, D. and Venugopal, S. "The Grid Economy", Special Issue on Grid Computing, Proceedings of the IEEE, Manish Parashar and Craig Lee (editors), Volume 93, Issue 3, 698-714pp, IEEE Press, New York, USA, March 2005. — 3-18 .. 3-34

(4) Buyya, R., Murshed, M, Abramson, D. and Venugopal, S. "Scheduling Parameter Sweep Applications on Global Grids: A Deadline and Budget Constrained Cost-Time Optimisation Algorithm", International Journal of Software: Practice and Experience (SPE), Volume 35, Issue 5, pp 491 - 512, Wiley Press, New York, USA, April 25, 2005. — 3-35 – 3-56

(5) Ayyub, S., Abramson, D., Enticott, C., Garic, S., Tan, J. "Fault-tolerant execution of large parameter sweep applications across multiple VOs with storage constraints", Concurrency and Computation: Practice and Experience. Currently Online, 25/8/2008. — 3-57 .. 3-72

Contributions

Giddy, Enticott and Garic were research associates under my supervision; Kotler provided a case study in X-Ray physics; Buyya was a PhD student under my supervision; Venugopal was Buyya's PhD student; Murshed provided expertise in discrete event simulation.

Scheduling Large Parametric Modelling Experiments on a Distributed Meta-computer

David Abramson† and Jon Giddy‡

† Department of Digital Systems,
Monash University,
Clayton,
VIC 3168.

‡ CRC for Distributed Systems Technology
Gehrmann Laboratories,
University of Queensland,
St Lucia, QLD.

Abstract

Nimrod is a tool which makes it easy to parallelise and distribute large computational experiments based on the exploration of a range of parameterised scenarios. Using Nimrod, it is possible to specify and generate a parametric experiment, and then control the execution of the code across distributed computers. Nimrod has been applied to a range of application areas, including Bioinformatics, Operations Research, Electronic CAD, Ecological Modelling and Computer Movies.

Nimrod was extremely successful at generating work, but it contained no mechanisms for scheduling the computation on the underlying resources. Consequently, users would not have any idea when an experiment might complete. We are currently building a new version of Nimrod, called Nimrod/G. Nimrod/G will integrate Nimrod job generation techniques with Globus, an international project which is building the underlying infrastructure for large meta-computing applications.

Using Globus, it will be possible for Nimrod users to specify time and cost constraints on computational experiments. Globus provides mechanisms for estimating execution time and waiting delays when using networked queued supercomputers. Nimrod/G will then use these to schedule the work in a way which meets user specified deadlines and cost budgets. In this way, multiple Nimrod users can obtain a quality-of-service from the computational network.

1. Introduction

Nimrod is a tool which makes it easy to parallelise and distribute large computational experiments based on the exploration of a range of parameterised scenarios [1, 2, 3, 12]. Using Nimrod, it is possible to specify and generate a parametric experiment, and then control the execution of the code across distributed computers. This mode of exploration is becoming increasingly important, as computational models are used instead of real-world experiments. For example, an engineer may use a computational to calculate the efficiency of an airfoil. It may be necessary to explore the behaviour of a wing by varying some key parameters and observing the result. Nimrod not only makes it easy to construct this type of experiment, but it also distributes the computation across idle workstations automatically. In this way, it is possible to run very large computational experiments and gain valuable insight into different design scenarios. Nimrod has been

applied to a range of application areas, including Bioinformatics, Operations Research, Electronic CAD, Ecological Modelling and Computer Movies. A commercial version, called "Clustor" is available [4].

Specifically, a Nimrod experiment consists of a computation and a collection of parameter domains, each of which is a set of possible values for a given parameter. Nimrod takes the cross product of all potential parameter values across each domain and executes the computation for every combination. Nimrod distributes work across distributed computers, but does not assume a shared file system. A simple, but powerful, scripting language makes it possible to describe details specific to the experiment, including file transfer and parameter substitution. More details of this scheme are discussed in [1] and [2].

Nimrod has been extremely successful at generating work because it takes the cross product of all possible parameter values. However, it contains no mechanisms for dynamically discovering suitable computing resources or for scheduling the computation on the underlying platforms. Specifically, Nimrod reads a file containing a list of machine names at startup, and only uses these machines for the duration of one experiment. The lack of scheduling means that a Nimrod user does not know when an experiment might complete. In particular, if several Nimrod experiments are run concurrently, then all of the experiments compete for computational servers on a random basis.

Interaction with Nimrod users has indicated that these two aspects are important, namely the ability to locate computational resources automatically, and also to give some indication of a likely completion time. This latter information may be used interactively by the user to alter the size of an experiment depending on the available computing power at that time. Accordingly, we are currently building a new version of Nimrod, called Nimrod/G. Nimrod/G will integrate Nimrod's job generation techniques with Globus, an international project which is building the underlying infrastructure for large meta-computing applications [7].

Nimrod/G will allow a user to specify a real time *deadline* for an experiment. From a user's perspective, it will be the obligation of the system to meet this deadline, or to report back that it is not possible given the available resources. Similarly, each of the underlying machines can be assigned a cost, which could be measured in units of currency (like dollars) or using virtual units such as a share in a distributed meta-computer. Accordingly, the user will be able to specify a cost constraint for an experiment, and the system will attempt to perform the work within the allocated budget.

Globus provides mechanisms for estimating execution time and waiting delays when using networked queued supercomputers. Nimrod/G will use these to schedule the work in a way which meets user specified deadlines and cost budgets. In this way, multiple Nimrod users can obtain a quality-of-service from the computational network.

Globus and Nimrod/G

"The Globus project is developing the basic infrastructure required to support computations that integrate geographically-distributed computational and information resources. Such computations may link tens or hundreds of resources located in multiple administrative domains and connected using networks of widely varying capabilities. Existing systems have only limited abilities for identifying and integrating new resources, and lack scalable mechanisms for authentication and privacy. Globus will contain a parallel programming environment that supports the dynamic identification and composition of resources available on national-scale internets, and mechanisms for authentication, authorization, and delegation of trust within environments of this scale." [11]

The goal of our current work is to create a version of Nimrod that allows the user to select time and cost constraints for the execution of their experiment within a meta-computing environment. Although time constraints cannot be guaranteed in a best-effort system, with suitable information, we can attempt to meet a user-supplied deadline within a given confidence level.

A major impediment to the development of such meta-computing projects is the difficulty of using heterogeneous systems in a scalable manner. The Globus toolkit overcomes this difficulty by providing a uniform interface to heterogeneous subsystems, while transparently using the best resources available. Nimrod/G uses Globus for its rich resource management functionality within a meta-computing infrastructure.

Table 1 – Globus-Nimrod/G mappings

communication	process creation	file transfer	security	resource location / inquiry	resource allocation / status
Nexus		RIO	GSSAPI and Access Control	Meta-computing Directory Service	Globus Resource Allocation Manager

Table 1 indicates the mapping between the functionality required by Nimrod/G and the Globus components. For each environment, these components are optimised to provide the local protocol most suitable for high performance meta-computing. For example, the Globus communication service (the Nexus module [8]) allows Nimrod to make use of high-speed network protocols where they are available, or the standard Internet Protocol, otherwise. Other services used by Nimrod include process creation, file transfer [10], security [9] and information services [6].

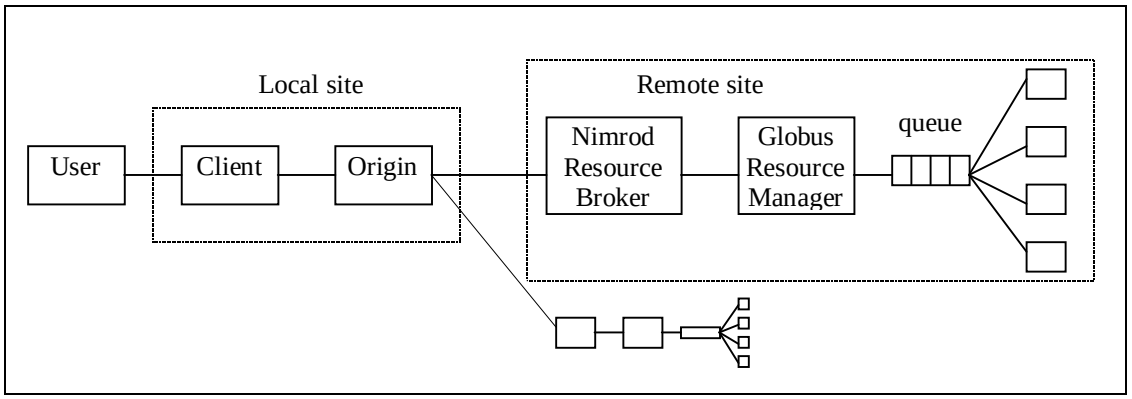
Globus provides a novel resource management module. In addition to generic interfaces for resource location and resource allocation in standalone, batch queueing, and real time systems, the module can provide an estimate of the expected starting time for a process. This is particularly useful within a batch queuing environment, currently the most

common setup on the supercomputer and cluster systems we intend using. This capability provides the basis for Nimrod's ability to satisfy the user's deadline.

Nimrod/G System Architecture

Figure 1 illustrates the Nimrod/G system architecture. The system is divided into a single local site and multiple remote sites. On the local site, the *origin process* operates as the

Figure 1 – Nimrod/G Architecture



master for the whole system. The origin process exists for the entire length of the experiment, and is ultimately responsible for the execution of the experiment within the time and cost constraints.

The user is either a person or process responsible for the creation of the experiment. The user interacts with the origin process through a client process. Neither the user nor the client are required to be available all the time. The distinction between the Client and the Origin is useful because a client may be tied to a particular display or environment. The user can stop the client, move to another environment and start another client, without affecting the origin process and thus the progress of the experiment. In addition, it is possible for multiple clients to monitor the same experiment by connecting to the one Origin process.

Each remote site consists of a cluster of computational nodes. A cluster may be a single multiprocessor machine, a cluster of workstations, or even a single processor. The defining characteristic of a cluster is that access to all nodes is provided through *a set of resource managers*, which are part of the Globus infrastructure. Each Globus Resource Manager (GRM) represents a method for accessing processing resources. Typically, the method is a queue in a batch queueing system, such as Condor [13] or LSF [14].

Before submitting any jobs to a cluster, the origin process uses the Globus process creation service to start a Nimrod Resource Broker (NRB) on the Cluster. The NRB is a different entity to the resource manager. It provides capabilities for file staging, creation of jobs from the generic experiment, and process control beyond that provided by the Globus Resource Manager.

The Client Process

The client process performs several tasks. First, the client assists the user in setting reasonable constraints. By using the Globus Metacomputing Directory Service, the client determines a probability of particular time and cost constraints being achieved for the user's experiment. This allows the user to determine a "quote" for execution of the experiment. The user may then decide to modify the constraints to achieve a particular outcome, trading a later deadline for a lower cost, or a higher cost for more chance of completion within the deadline.

Second, once the user is satisfied with a particular combination of constraints, the client sends the experiment to the origin process along with the deadline and cost constraints. In effect, this begins the execution of the experiment.

The final task of the client is to provide feedback to the user about the progress of the experiment. This information is obtained from the origin process, which as the master of the whole experiment, must keep status on the progress of the experiment.

The Origin Process

The origin process creates a pool of jobs, one job for each unique combination of parameter values. It obtains a set of clusters of computational nodes from the Globus MDS. Selection of clusters is based on cost, communication capacity, and computational capacity available before the deadline. Total cost is an absolute limit imposed by the user. The other two criteria attempt to minimise the total communication delay. Communication capacity is used to select topologically "nearer" clusters. A cluster with more computational capacity reduces the need to split the experiment into smaller sub-experiments. Because each cluster need only fetch the input data once, it may be more efficient to send 20 jobs to a more "distant" cluster than to send 10 jobs to each of two nearer clusters.

For each cluster, the origin process starts a local handler thread. The first task of this thread is to start a Nimrod Resource Broker on its cluster through the Globus process creation service (part of the Nexus module). It then interrogates the cluster for an up-to-date estimate of its computational capacity. This estimate is determined heuristically from information provided by the Globus resource manager and any other NRB's executing on the cluster. Once a thread has received the estimate from its cluster, it removes a number of jobs from the common pool. The number of jobs is the number required to fill the cluster up to the deadline. Using multiple threads, it is possible to mask the latency introduced by communicating with distant or slow clusters.

If there is more computational capacity than required, some clusters will be busy until the deadline, while others may be idle. This is almost the opposite of load balancing. Instead of maintaining similar loads on each cluster, a variety of load patterns are maintained on each cluster. This is called load profiling [5]. It is useful for our problem because the goal is not to reduce individual job response times, but rather for all jobs to meet the deadline. The less loaded clusters provide an overflow resource for handling

variances in system behaviour. These variances occur because the resource managers typically provide best-effort service only, and other users are competing for the computational resource.

If there is less computational capacity than required some jobs will remain in the pool. Further, additional jobs from over-committed clusters will arrive in the pool. Periodically, if jobs still exist in the pool, the origin process obtains more clusters from the MDS. These new clusters compete with any lightly loaded clusters for the remaining jobs.

The Nimrod Resource Broker

The Nimrod Resource Broker ensures that all jobs it handles complete by their deadline. It monitors the probability of successful completion of all jobs it is locally responsible for using information supplied by the Globus Resource Manager.

A job submitted to a NRB is not necessarily submitted immediately to the local Resource Manager. Reasons for delayed submission include reaching a queue submission limit. The NRB may determine that there is adequate slack time (the difference between the estimated completion time and the deadline) for the job to be completed (within a given probability). For this reason, any slack time is kept by the NRB to handle delays from preemption by higher priority queues or from longer than expected execution times for preceding jobs.

If the probability of successful completion of a job on a particular cluster falls below a threshold, the origin removes jobs from the NRB until the probability of successful completion of the remaining jobs is above the threshold. The origin process returns jobs to the common pool, where they are redistributed, either to clusters performing ahead of schedule, or to new clusters.

Conclusion

This paper has outlined the key architectural issues in the design of Nimrod/G. At this stage there are many important details which still need to be specified.

The first stage of the development will involve interfacing Nimrod to the basic services provided by Globus. The new architecture, along with the Globus services, will provide more scalability and compatibility over a wide-area environment than was previously available under Nimrod. Nimrod/G is actually being constructed using some components from the commercial version of Nimrod, “Clustor”. Clustor is decomposed into a number of separate programs, each responsible for part of the process of managing an experiment. The first two phases, which are responsible for creating a control script and generating an experiment parameter set, are controlled by two programs called the “preparator” and the “generator” [4]. These functions can be used without modification. The final phase, job dispatch, is being rewritten to make use of Globus.

Initially, the system will attempt to estimate the completion time for each cluster, and indicate to the user whether the experiment will complete within the deadline. Of course,

the statistics for the estimated starting times are only approximate, and thus Nimrod/G cannot guarantee that the deadline will be met. Accordingly, the next phase of the implementation will involve building a dynamic job redistribution facility. In this scheme, jobs will be moved from one cluster to another depending on advice from the Nimrod Resource Brokers as to whether the cluster is like to meet the deadline for the currently allocated jobs. If a cluster is unable to meet a deadline, the jobs are returned to the Origin and redistributed. This phase will require the development of some heuristics for deciding when a job should be relocated prior to its execution.

Acknowledgements

This project is funded by the Co-operative Research Centre for Distributed Systems Technology.

References

[1] Abramson , D., Sosic , R., Giddy , J., Cope , M. "The Laboratory Bench: Distributed Computing for Parametised Simulations", 1994 Parallel Computing and Transputers Conference, Wollongong, Nov 94, pp 17 - 27.

[2] Abramson D., Sosic R., Giddy J. and Hall B., "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[3] Abramson, D., Foster, I., Giddy, J., Lewis, A., Sosic, R., Sutherst, R. and White, N. "Nimrod Computational Workbench: A Case Study in Desktop Metacomputing", Australian Computer Science Conference (ACSC 97), Macquarie University, Sydney, Feb 1997.

[4] Active Tools Corporation, http://www.activetools.com

[5] Bestavros, A. "Load profiling: A methodology for scheduling real-time tasks in a distributed system." In Proceedings of ICDCS'97: The IEEE International Conference on Distributed Computing Systems, Baltimore, Maryland, May 1997.

[6] Fitzgerald, S., Foster, I., Kesselman, C., von Laszewski, G., Smith, W. and Tuecke, S. "A directory service for configuring high-performance distributed computations." Proceedings 6th IEEE Symposium on High Performance Distributed Computing. To appear.

[7] Foster I. and Kesselman, C. "Globus: A metacomputing infrastructure toolkit." International Journal of Supercomputer Applications, 1997. To appear.

[8] Foster, I., Geisler, J., Kesselman, C. and Tuecke, S. "Managing multiple communication methods in high-performance networked computing systems." Journal of Parallel and Distributed Computing, 40:35-48, 1997.

[9] Foster, I., Karonis, N.T., Kesselman, C., Koenig, G. and Tuecke, S. "A secure communications infrastructure for high-performance distributed computing." Proceedings 6th IEEE Symposium on High Performance Distributed Computing. To appear.

[10] Foster, I., Kohr, Jr., D., Krishnaiyer, R. and Mogill, J. "Remote I/O: Fast access to distant storage." Proceedings IOPADS'97. To appear.

[11] Globus Project, http://www.globus.org/globus/information.htm

[12] Lewis, A., Abramson D., Sosic R., Giddy J., "Tool-based Parameterisation : An Application Perspective", Computational Techniques and Applications Conference, Melbourne, July 1995.

[13] Litzkow, M., Livny, M. and Mutka, M.W., "Condor – A Hunter of Idle Workstations", proceedings of the 8th International Conference of Distributed Computing Systems, pp 104-111, June 1988.

[14] Zhou, S., Zheng, X., Wang, J. and Delisle, P. "Utopia: A load sharing facility for large, heterogeneous distributed computer systems", Software-Practice and Experience 23(12): 1305-1336, December 1993.

High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?

David Abramson †, Jon Giddy‡ and Lew Kotler §

† School of Computer Science and Software Engineering
Monash University,
CLAYTON, VIC 3168, Australia

‡ CRC for Distributed Systems Technology
General Purpose South Building,
University of Queensland,
ST LUCIA, QLD, Australia

§ Australian Radiation Protection and Nuclear Safety Agency
Lower Plenty Road
YALLAMBIE VIC 3085, Australia

Abstract

This paper examines the role of parametric modeling as an application for the global computing grid, and explores some heuristics which make it possible to specify soft real time deadlines for larger computational experiments. We demonstrate the scheme with a case study utilizing the Globus toolkit running on the GUSTO testbed.

1 Introduction

Parametric computational experiments are becoming increasingly important in science and engineering as a means of exploring the behavior of complex systems. For example, an engineer may explore the behaviour of a wing by running a computational model of the airfoil multiple times while varying key parameters such as angle of attack, air speed, etc. The results of these multiple experiments yield a picture of how the wing behaves in different parts of parametric space.

Over the past several years, we have developed a specialized parametric modeling system called Nimrod [1][2][3][17]. Nimrod uses a simple *declarative* parametric modeling language to express a parametric experiment and provides machinery that automates the task of formulating, running, monitoring, and collating the results from the multiple individual experiments. Equally important, Nimrod incorporates a distributed scheduling component that can manage the scheduling of individual experiments to idle computers in a local area network. Together, these features mean that even complex parametric experiments can be defined and run with little programmer effort. In many cases it is possible to establish a new experiment in minutes. Nimrod has been applied to a range of application areas, including Bioinformatics, Operations Research, Network Simulation, Electronic CAD, Ecological Modelling and Business Process Simulation [14][17].

Whilst Nimrod has been very successful, it suffers from a few limitations when considered in the context of a "Global Computational Grid", in which a large number of computers are linked globally to form a seamless supercomputer [12]. First, it uses a static set of resources and does not "discover" new ones dynamically. This makes it more suited to fixed infrastructure than the type found on a global grid. Second, it has no idea of user deadlines. This is adequate when there are only a few users and a fixed resource, in which case, a user can estimate the time that a computation will take to complete, and adjust the experiment to meet project milestones. However, in a global grid, it is very difficult for the user to compute when a computation may complete because the underlying resources provide no performance guarantees, and Nimrod does not provide any mechanism to allow a user to specify such a deadline. Such deadline control, thus, becomes important as Nimrod is deployed in a global grid because the timeliness of a solution may be very important to the end user. Third, Nimrod relies on standard Unix level security. In a global grid, the owners of expensive supercomputing facilities require more elaborate security mechanisms. Finally, Nimrod does not support a range of access mechanisms, such as those provided by various queue managers. In the global grid many different queue managers co-exist and must be supported, since no single solution will be acceptable.

In this paper, we describe a new version of Nimrod, called Nimrod/G, which addresses these shortcomings. A particular focus of this paper is how the basic Nimrod model can be extended to provide (soft) performance guarantees in a dynamic and heterogeneous "Computational Grid". We

describe a simple but effective scheduling component in Nimrod/G which seeks to meet such constraints by a dynamic and iterative process of resource discovery, resource acquisition, and resource monitoring.

The paper introduces Nimrod and Globus, the toolkit on which Nimrod/G is built. It then describes the scheduling algorithm, and illustrates the utility of the system by a case study.

2 Parametric Modeling with Nimrod and Clustor

Nimrod is a tool that manages the execution of parametric studies across distributed computers. It takes responsibility for the overall management of an experiment, as well as the low-level issues of distributing files to remote systems, performing the remote computation and gathering the results. Clustor [4] is a commercial version of the research system Nimrod.

When a user describes an experiment to Nimrod, they develop a *declarative* "plan" file which describes the parameters, their default values, and the commands necessary for performing the work. The system then uses this information to transport the necessary files and schedule the work on the first available machine.

A plan file is composed of two main sections, the parameter section and the tasks section. Figure 1 shows a sample plan used for some of the simulation work discussed in this paper. The experiment consists of varying the `thickness` parameter and each execution receives a different seed value. Nimrod generates one job for each unique combination of the parameter values, by taking the cross product of all values. In the plan in Figure 1, 400 jobs would be generated to control the computational physics case study.

When the user invokes Nimrod on a workstation, the machine becomes known as the "root" machine because it controls the experiment. When the dispatcher executes code on remote platforms, each of these is known as a computational "node". Thus a given experiment can be conducted with one root and multiple nodes, each of a different architecture if required.

Nimrod supports five phases of a computational experiment. Phases 1 and 5 are performed once per experiment, while Phases 2, 3 and 4 are run for each distinct parameter set.

1. Experiment pre-processing, when data is set up for the experiment;
2. Execution pre-processing, when data is prepared for a particular execution;
3. Execution, when the program is executed for a given set of parameter values;
4. Execution post-processing, when data from a particular execution is reduced; and
5. Experiment post-processing, when results are processed, for example by running data interpretation or visualization software.

```
parameter iseed integer range from 100
to 4000 step 100;
parameter thick label "BUC thickness"
float range from 1.1
      to 2.0 step 0.1;
parameter jseed integer compute
thick*1000;

task nodestart
      copy ccal.$OS node:./ccal
      copy dummy node:.
      copy ccal.dat node:.
      copy skel.inp node:.
endtask
task main
      node:substitute skel.inp ccal.inp
      node:execute ./ccal
      copy node:ccal.op ccalout.$jobname
endtask
```

Figure 1 – Sample Plan file

During phases 2 and 4 files may be moved between the root machine and the cluster processes – unlike general parallel computing this is the only communication that occurs between tasks.

In this example there are two tasks – "main" and "nodestart". The "main" task is executed for each set of parameters. It runs a simulation, called `ccal`, on a node, passing the parameter values to the program via a parameter file. It then copies a number of result files back to the root machine, appending each name with a unique identifier. This task corresponds to the 3rd phase discussed in the previous section. The "nodestart" task executes once per experiment, and corresponds to the 1st phase described in the previous section. It copies a file to the remote node which is common across all simulations, and a skeleton of the input parameter file. The latter is processed using the substitute command, which replaces placeholders with actual values. It also copies the correct binary (ccal) for the target operating system.

The plan file is processed by a tool called the "generator". The generator takes the parameter values, and gives the user the choice of actual values. It then builds a "run" file, which contains a description for each job. The "run" file is processed by another tool called the "dispatcher", which is responsible for managing the computation across the nodes.

The dispatcher implements the file transfer commands, as well as the execution of the model on the remote node. In Nimrod and Clustor, the dispatcher allocates work to machines without any attempt to schedule their execution.

Nimrod/G, which is described in more detail in Section 4, makes use of two components of Clustor, namely the preparator and the generator. The dispatcher has been completely rewritten to support scheduling and make use of the Globus toolkit.

3 Globus

3.1 Toolkit functionality

The Globus Grid toolkit is a collection of software components designed to support the development of applications for high-performance distributed computing environments, or "Grids" [12]. The Globus toolkit is an implementation of a "bag of services" architecture, which provides application and tool developers not with a monolithic system but rather with a set of standalone services. Each Globus component provides a basic service, such as authentication, resource allocation, information, communication, fault detection, and remote data access [11][13][8]. Different applications and tools can combine these services in different ways to construct "Grid-enabled" systems. Four such components are used in Nimrod/G:

- The Globus Resource Allocation Manager (GRAM) service provides an API for requesting that computations be started on a computational resource, and for managing those computations once they are started.

- The Metacomputing Directory Service (MDS) provides an API for discovering the structure and state (e.g., availability) of resources that may be of interest to a computation.

- The Globus Security Infrastructure (GSI) provides single sign-on, run anywhere (where authorized) capabilities for computations such as ours that need to operate on many computers.

- The Global Access to Secondary Storage (GASS) service provides uniform access mechanisms and APIs for files stored on various storage systems.

The Globus resource management architecture is currently being extended to support advance reservations and to support differentiated services networks. Experiments with the advance reservation of differentiated services networks are being conducted in the context of both a local area testbed at Argonne and over an ESnet-based testbed linking ANL and LBNL. We anticipate this work being useful future versions of Nimrod, as a means of ensuring that computations can be performed on a required schedule.

3.2 Scheduling on the Grid

Scheduling programs on parallel and distributed computers has been the topic of research for many years, and a great deal has been accomplished. Much of the work has been targeted at scheduling and placing processes on a parallel computer in which the program can be represented as a task graph and the amount and nature of the computational resource is static for the duration of the execution. This environment leads naturally to scheduling heuristics, which try and produce a schedule that observes a number of constraints. Often there is a goal to minimise the execution time of the task or spread the load optimally.

The Computational Grid presents a much more challenging domain for scheduling because the performance of the underlying resource is highly variable, and can change even once an application has been scheduled. Accordingly, it is not possible to consider the Grid as a single computer system under the control of a single scheduler. The application domain of parameter studies also places additional demands on the scheduler. Such studies are often complex and very time consuming, and it is important to receive the results in a timely manner. Where traditional parallel computing scheduling systems may attempt to minimise the execution time of an application, or optimise the load distribution across the machines, we argue that allowing the user to specify an absolute (but soft) deadline is also a useful way of expressing the timeliness of the computation. Interestingly, deadlines have been commonly applied in real-time systems as the basis for scheduling.

Nimrod/G is a "Grid aware" application. It exploits an understanding of its problem domain as well as the nature of the computational Grid to provide a high level interface to the user. Specifically, it provides transparent access to the computational resources, and implements user level scheduling. In our case, we attempt to schedule otherwise unrelated tasks so that a user-supplied deadline is met. In the basic Nimrod model there is no communication between tasks once they have started, and thus the scheduling problem becomes one of finding suitable resources and executing the application. Although the problem sounds simple, the dimension of scheduling complexity is increased due to the introduction of parameters such as computational economics, deadlines, the usage of resources scattered over different administrative domains and varied machine and communication performance. A number of other researchers are also considering the scheduling problem in the context of their own applications, including Ninf [21], AppLeS project

[6][5], NetSolve [9], Nile [20] and NEOS [10]. In Condor [19] matchmaking is used to match resources with tasks, but parameter studies are not supported explicitly, as here, and no deadline scheduling is performed.

4 Nimrod/G Description

4.1 Architecture

Nimrod/G is designed to operate in an environment that comprises a set of sites, each providing access to a set of computers with their own administrative control, with access to these resources being mediated by a Globus Resource Allocation Manager (GRAM). Information about the physical characteristics and availability of these and other resources are available from the Globus directory service (MDS).

As illustrated in Figure 2, a user (or process acting on their behalf) initiates a parametric study at a *local site*; Nimrod/G then organizes the mapping of individual computations to appropriate *remote sites*, according to the Nimrod/G scheduling heuristic. On the local site, the *origin process* operates as the master for the whole system. The scheduling and monitoring logic described below is encapsulated in an *origin process*, which exists for the entire length of the experiment and is ultimately responsible for the execution of the experiment within the specified time and cost constraints. We note that this structure differs significantly from that used in Nimrod and Clustor as described in Section 2, because it has been designed to support the scheduling of computations on resources scattered across the globe with their own administrative policy and control.

The user is either a person or process responsible for the creation of the experiment. The user interacts with the origin process through a client process. Neither the user nor the client are required to be available all the time. The distinction between the Client and the Origin is useful because a client may be tied to a particular display or environment. The user can stop the client, move to another environment and start another client, without affecting the origin process and thus the progress of the experiment. In addition, it is possible for multiple clients to monitor the same experiment by connecting to the one Origin process. We have used this feature to allow a team in Melbourne and one in Chicago to observe the same computational experiment. A sample GUI for this monitor is shown in Figure 3. This example shows a number of jobs running on various GUSTO resources. It also shows the deadline and cost budget interface.

Each remote site consists of a *cluster* of computational nodes. A cluster may be a single multiprocessor machine, a cluster of workstations, or even a single processor. The defining characteristic of a cluster is that access to all nodes is provided through *a set of resource managers* provided by the Globus infrastructure. Each Globus Resource Allocation Manager (GRAM) implements a method for accessing processing resources. Typically, the method is a queue in a batch queuing system, such as Condor [18] or LSF [16] [23] or a process fork on a shared memory machine.

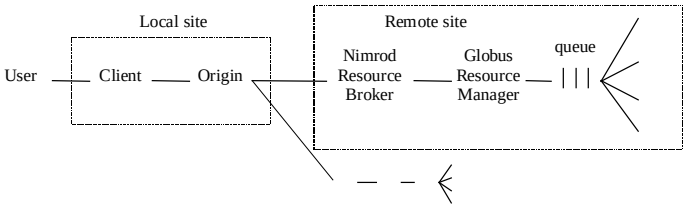


Figure 2 – Nimrod/G Architecture



Figure 3 – Sample monitor GUI

Before submitting any jobs to a cluster, the origin process uses the Globus process creation service to start a Nimrod Resource Broker (NRB) on the Cluster. The NRB is a different entity to the resource manager. It provides capabilities for file staging, creation of jobs from the generic experiment, and process control beyond that provided by the GRAM.

4.2 Cost in a Global Grid

Unless restrictions are placed on access to the various resources of a global grid, it is likely to become congested with too much work in short order. We believe a suitable model for controlling the amount of work requested is a fiscal one in which users pay for access. Such a scheme allows resource providers to set pricing rates for the various

machines, which can vary between classes of machine, times of the day, resource demand and classes of user. Even communication cost can be taken into consideration especially when data communication cost is significant compared to computation.

Whilst we have not developed a complete model of cost, and how it can be used in a global grid, we have experimented in Nimrod/G with defining a cost matrix between users and resources. For example, in Figure 4 the cost of resources 1 and 4 is different for users 1 and 4. In particular, user 1 finds resource 1 cheaper than 4, but user 4 find resource 4 cheaper than 1. This asymmetry means that the initial resources for each user may be different, providing for example a mechanism for favoring local machines first. At present we generate the cost matrix by hand based on a mutual agreement on the relative costs of systems. In the future, we expect that a more elaborate costing scheme will need to be devised, and this is a topic for ongoing research. In the next section we describe how cost is used in the current version of Nimrod/G.

		Resources				
		1	2	3	4	5
Users	1	1			3	
	2					
	3					
	4	2			1	
	5					

Figure 4 – Sample Nimrod/G Cost Matrix

4.3 Scheduling Algorithm

The Nimrod/G scheduler is responsible for discovering and allocating the resources required to complete an experiment, subject to specified execution time and budget constraints. This scheduling problem is made more complicated by the fact that in a Grid environment, we typically cannot guarantee exclusive or immediate access to underlying resources. Accordingly, *traditional* scheduling techniques (as discussed in [5]) are not necessarily applicable, as both resource availability and execution rates can vary unpredictably.

We address this problem by the use of the following simple heuristic:

1. (Discovery) First the number and then the identity of the lowest-cost set of resources able to meet the deadline are identified. A cost matrix (as described in Section 4.2) is used to identify low-cost resources; queries to the MDS directory service are then used to determine resource. The output from this phase is a set of resources to which jobs should be submitted, ordered by the cost to *this* user. Different users may, therefore, generate lists sorted in a different order.

2. (Allocation) Unscheduled jobs (maintained in a pool, one per parameter set) are allocated to the candidate resources identified in Step 1.

3. (Monitoring) The completion time of submitted jobs is monitored, hence establishing an *execution rate* for each resource.

4. (Refinement) Rate information is used to update estimates of typical execution times on different resources and hence the expected completion time of the job. This refinement process may lead us to return to Steps 1 or 2 to discover new resources or to drop existing resources from the candidate set.

The scheme continues until the deadline is met, or the cost budget is exceeded. If the latter occurs, the user is advised and the deadline can be modified accordingly. One consequence on the way we have chosen to use cost is that the cost of an experiment will vary depending on the load and the profile of the users at that time. This reflects the type of pricing that occurs in an auction – less demand will allow the experiment to be performed on cheaper resources.

As we shall see below, initial experiments suggest that this heuristic works well in practice: applications tend to determine quite quickly what is an appropriate job submission rate, and use expensive resources only when necessary.

5 A Case Study

We report on experiment that we have conducted to evaluate the effectiveness of the Nimrod/G architecture and scheduling heuristics in a real application. The experiment was using resources provided by sites participating in the international Globus Ubiquitous Supercomputing Testbed Organization (GUSTO). GUSTO sites run standard Globus software and can sometimes be persuaded to make their resources available for Grid computing experiments. As show in Table 1, the study reported here used resources at Argonne, Boston University, Monash University, Northern Illinois University, USC Information Sciences Institute, and the University of Wisconsin. These resources are quite diverse in terms of their size, availability, architecture, processing capability, power, performance, scheduling mechanism (immediate access or “fork”, Condor-based queue, LSF-based queue), and geographic location. The “Max number nodes” column in Table 1 indicates the number of nodes in the machine, while “Average free nodes”

indicates the average number available during the experimental period in April 1999. The table also shows a (currently artificial) cost value for each system.

5.1 Ionization Chamber Calibration

The Australian Radiation Protection and Nuclear Safety Agency (ARPANSA) provides the Primary Australian Standards for certain Radiological quantities such as Air Kerma, which essentially quantifies photon energy deposition in terms of ionization in air. The calibration of medical equipment used for diagnosis and/or treatment against these standards ensures that in any location the radiation doses received by patients is consistent with that received in other locations in Australia and overseas.

An ionization chamber essentially isolates a certain volume of air and measures the ionization within that volume. However, the physical process of isolating a volume of air modifies the original photon and electron spectrum entering the volume. Thus, if the chamber is to act as a primary standard for calibration purposes, it is necessary to correct the measured ionization for the spectral perturbations due to the physical presence of the chamber.

The nature of the problem is such that not all correction factors can be determined experimentally. Within the field of radiation dosimetry, Monte-Carlo simulation programs, such as the EGS4 package [22], have become an acceptable method of simulating the response of ionization chambers to photons and electrons and thus an alternative means of determining the appropriate correction factors. The absolute accuracy of the EGS4 package for simulating ion chambers is quoted at about 1% but relative accuracy is actually better than this.

One important correction factor for the primary standard, in the standardization of photons emitted by a cobalt-60 teletherapy source, is the extent of photon attenuation and scattering in the front wall of the chamber. Detailed measurements of chamber response as a function of front wall thickness have been performed and a quantitative model of interactions within the front wall used to determine the appropriate correction factors. These detailed measurements provide an interesting basis for comparison with Monte-Carlo, as not only the chamber but also the cobalt-60 photon spectrum are to be simulated. The calculations reported here concern the simulation of the chamber response as a function of front wall thickness. We used Nimrod/G to perform this parametric variation whilst also running the model many times with different random number seeds and different front wall thicknesses. As the calculated data may not be normally distributed, it was necessary to calculate the averages of sets of data and to take the mean of the averages to obtain a mean value (whose components are normally distributed) for any particular front wall thickness.

5.2 Scientific Results

One of the most important output variables of the model is the number of ion pairs created in the collecting volume per incident photon. Accordingly, we wish to compare the simulation data with experimental data. In order to do this, we fit both the experimental and simulation results to an expression of the form $Ae^{-\mu t} + Be^{-\beta t}$, where μ is the effective attenuation coefficient for photons, β is the effective attenuation coefficient for electrons and t is the thickness of the front wall of the chamber. A comparison for the fitted coefficients for both the simulation and experiment is given in Table 2. These results show that the EGS4 model is quite accurate. Further, by considering the data as plotted against thickness in Graph 1, the model is particularly accurate when the thickness is less than 0.4 cm.

In order to improve the accuracy of the model, we believe that we require in the order of 5 times the number of simulations. Given that the results could be obtained on a relatively modest number of processors in the order of a working day, this should be achievable easily. We plan to perform these simulations in the near future.

5.3 Computational Results

The simulations described were run on the subset of the GUSTO testbed resources defined in Table 1. Of the 364 nodes in this subset, about 70 were available to us during the trial. This number was sufficient to allow testing of many of the key features of Nimrod/G, in particular its ability to schedule tasks according to time and cost constraints.

The ionization chamber study involved 400 tasks. The execution time of the model varied depending on the platform used, ranging from about 45 minutes per parameter set on the SGI R10000 based machines to 140 minutes on the NIU and ISI Suns. Three separate experiments were performed with deadlines of 20 hours, 15 hours, and 10 hours, respectively, so as to allow an evaluation of Nimrod/G's ability to meet soft real time deadlines. Graph 2 shows the number of nodes used as a function of time for each deadline. Not surprisingly, Nimrod/G allocates additional resources for the more stringent deadlines. Whilst this result appears obvious, it should be stressed that these results have been obtained from a *real* test-bed, in which the resources are *discovered* and the load has been distributed *dynamically*. The algorithm has adapted the distribution of tasks without any prior knowledge of the initial load, test-bed configuration or speed of the individual machines.

Machine	Loca-tion	Method of starting Jobs/ Machine Type	Rel. Cost	Max nodes	Typ. avail
Pitcairn	ANL Chicago	Fork UltraSPARC-II	5	8	3
Tuva	ANL Chicago	Fork UltraSPARC-II	5	1	1
Goshen	U. Wisco	Condor Sun 4	5	175	7
Lemon	ANL Chicago	Fork MIPS R10000	10	2	2
Yukon	ANL Chicago	Fork MIPS R10000	10	16	16
Flash	ISI LA	Fork MIPS R10000	15	2	2
Jupiter	ISI LA	Fork MIPS R10000	15	8	8
Olympus	North Illinois	Fork UltraSPARC	15	1	1
Bolas	ISI LA	Fork UltraSPARC	20	1	1
Hammie	ISI LA	Fork UltraSPARC	20	1	1
Huntsman	ISI LA	Fork UltraSPARC	20	1	1
Hathor	Monash Australi a	Condor Pentium II/333	50	20	1
Denali	ANL Chicago	Fork MIPS R10000	50	96	24
Lego	Boston U	LSF MIPS R10000	50	32	6
Total				364	70

Table 1 – GUSTO machines used for case studies.

parameter	EGS4 calculation	experiment
μ	0.103	0.113
β	26.61	29.90
Ratio (B/A)	0.377	0.338

Table 2- Attentuation coefficients

Of more interest are Graphs 3, 4 and 5, which break down selected nodes according to cost, against time. These graphs show that the scheduler initially favours cheaper resources, but then, as time advances and it finds that the deadline cannot be achieved using the current resource set, more expensive machines are introduced. For example, in Graph 3 the 10 "cost unit" machines are introduced after about half an hour when the scheduler calculates that it cannot meet the 20 hour deadline using only 5 "cost unit" machines. Similarly, in Graph 5, it can be seen that 50 "cost unit" machines are introduced at around 2 hours in order to meet the 10 hour deadline, whilst these were not required in the 20 hour experiment. These predictions are based on the measurements of the real performance of the machines whilst running the ionization code.

Table 3 quantifies the impact on cost of the different node selections made for different deadlines: a 10 hour deadline costs more than three times a 30 hour deadline. This is a reflection of the distribution of resources available at the time we performed the work, and the particular cost matrix which was used. While we cannot easily show in such a dynamic environment that Nimrod/G is making "optimal" selections, it is clear that Nimrod/G is being effective in selecting more expensive nodes only when the system requires them to meet the deadline. Of course, these costs will vary depending on the current load on the test-bed as the lowest cost resources may not always be available.

Experiment	**Total Cost in Cost Units**
10 hours	9060
15 hours	5675
20 hours	2980

Table 3 – Cost of experiments

6 Conclusion

In this paper we have discussed the evolution of a particular tool, Nimrod, from a local computing environment to the Global Computational Grid. The paper has described the various services of Globus and how these have been applied in building a Grid aware application. In particular, we have focussed on the problem of providing some soft real time deadlines in such an environment. The algorithm used is both simple and adaptive to changes in the workload distribution on the grid, and incorporates user requirements as well as system ones. The case study has illustrated that it is possible to build an application which takes account of the highly dynamic and unpredictable nature of the grid.

However, we regard our efforts in this area as preliminary. We need to develop a more complete model of cost in such an environment, in particular issues such as storage of money and how it can be deployed. We wish to take account of the ability in Globus to reserve some resources, and incorporate this into the scheduling mechanism. It would appear that some notion of privilege or priority in addition to money would be important, particularly when a user wishes to make use of local resources, but also place them in the global grid. For example, under the current scheduling algorithm, it is possible for a user to be forced to use external, more expensive, resources just because the local ones are already working for another user. Finally, there are a number of implementation issues which need to be addressed, such as the way that Nimrod/G explores the Globus MDS in search of suitable machines. Such issues are the topic of ongoing research.

Acknowledgements

This project is funded by the Distributed Systems Technology Centre under the Australian Government CRC program. We wish to thank our colleague, Ian Foster, for his valuable contribution and support for this work, and Rajkumar Buyya for proofreading the paper. We would also like to acknowledge Stuart Martin and Miron Livny for their assistance in obtaining access to GUSTO and Condor resources.

References

[1] Abramson , D., Sosic , R., Giddy , J., Cope , M. "The Laboratory Bench: Distributed Computing for Parametised Simulations", 1994 Parallel Computing and Transputers Conference, Wollongong, Nov 94, pp 17 - 27.

[2] Abramson D., Sosic R., Giddy J. and Hall B., "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[3] Abramson, D., Foster, I., Giddy, J., Lewis, A., Sosic, R., Sutherst, R. and White, N. "Nimrod Computational Workbench: A Case Study in Desktop Metacomputing", Australian Computer Science Conference (ACSC 97), Macquarie University, Sydney, Feb 1997.

[4] Active Tools Corporation, http://www.activetools.com

[5] Berman, F. "High Performance Schedulers", In The Grid: Blueprint for a Future Computing Infrastructure, pages 279 - 309. Morgan Kaufmann Publishers, 1999

[6] Berman, F. and Wolski, R. "The AppLeS project: A status report", In proc. NEC Symposium on Metacomputing, 1997.

[7] Bestavros, A. "Load profiling: A methodology for scheduling real-time tasks in a distributed system." In Proceedings of ICDCS'97: The IEEE International Conference on Distributed Computing Systems, Baltimore, Maryland, May 1997.

[8] Bester, J., Foster, I., Kesselman, C., Tedesco, J. and Tuecke, S. "GASS: A Data Movement and Access Service for Wide Area Computing Systems", Proc. IOPADS'99, ACM Press, 1999.

[9] Casanova, H. and Dongarra, J. "NetSolve: A Network Server for Solving Computational Science Problems", The International Journal of Supercomputing Applications and High Performance Computing, Vol 11, Number 3, pp 212-223, 1997.

[10] Czyzyk, J., Mesnier, M. and More, J. "The Network-Enabled Optimization System (NEOS) Server", Preprint MCS-P615-0996, Argonne National Laboratory, Argonne, IL, 1996.

[11] Fitzgerald, S., Foster, I., Kesselman, C., von Laszewski, G., Smith, W. and Tuecke, S. "A directory service for configuring high-performance distributed computations.", Proceedings 6th IEEE Symposium on High Performance Distributed Computing, , 365-375, IEEE Press**,** 1997.

[12] Foster, I. and Kesselman, C. Globus: A Toolkit-Based Grid Architecture. In The Grid: Blueprint for a Future Computing Infrastructure, pages 259-278. Morgan Kaufmann Publishers, 1999.

[13] Foster, I., Kesselman, C., Tsudik, G. and Tuecke, S. "A Security Architecture for Computational Grids", ACM Conference on Computers and Security, 83-91, ACM Press, 1998.

[14] http://hathor.cs.monash.edu.au/RJK

[15] http://www.globus.org/

[16] http://www.platform.com

[17] Lewis, A., Abramson D., Sosic R., Giddy J., "Tool-based Parameterisation : An Application Perspective", Computational Techniques and Applications Conference, Melbourne, July 1995.

[18] Litzkow, M., Livny, M. and Mutka, M.W., "Condor – A Hunter of Idle Workstations", proceedings of the 8th International Conference of Distributed Computing Systems, pp 104-111, June 1988.

[19] Livny, M. High-Throughput Resource Management. In The Grid: Blueprint for a Future Computing Infrastructure, pages 311-337. Morgan Kaufmann Publishers, 1999

[20] Marzullo, K., Ogg, M., Ricciardi, A., Amoroso, A., Calkins, F. and Rothfus, E. "NILE: Wide area computing for high energy physics", proc. 1996 SIGOPS Conf. New York, ACM, 1996.

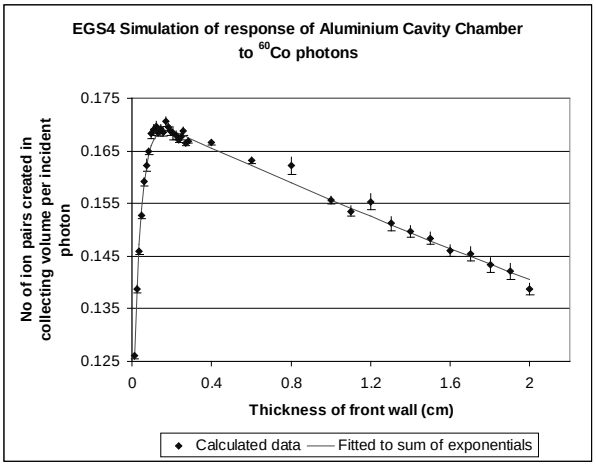
[21] Matsuoka, A., Nagashima, U. and Nakada, H. "Ninf: Network based information library for globally high performance computing", in Methods and Applications (POOMA), 1996.

[22] Nelson, W.R., Hirayama H. and Rogers, D.W.O. "The EGS4 Code System", Stanford Linear Accelerator Center Report SLAC-265 (1985)

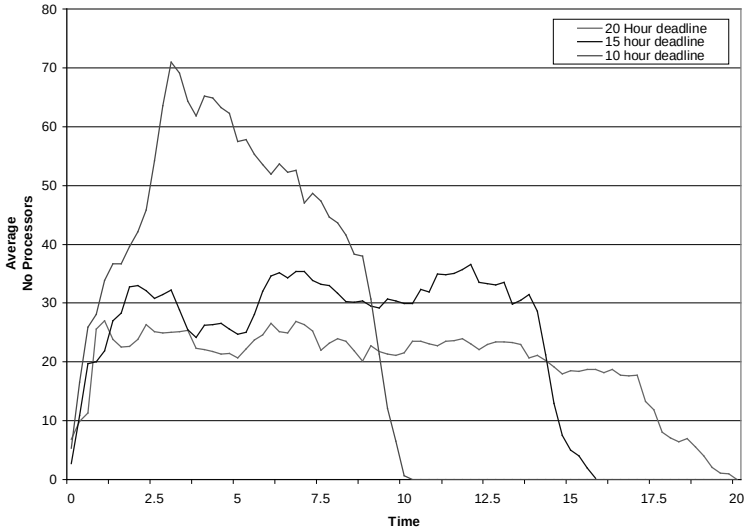
[23] Zhou, S., Zheng, X., Wang, J. and Delisle, P. "Utopia: A load sharing facility for large,

heterogeneous distributed computer systems", Software-Practice and Experience, 23(12): 1305-1336, December 1993.

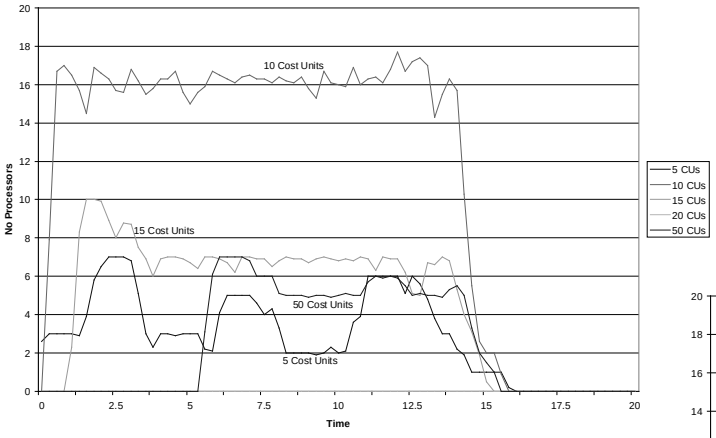
Graph 1 – Chamber response against thickness



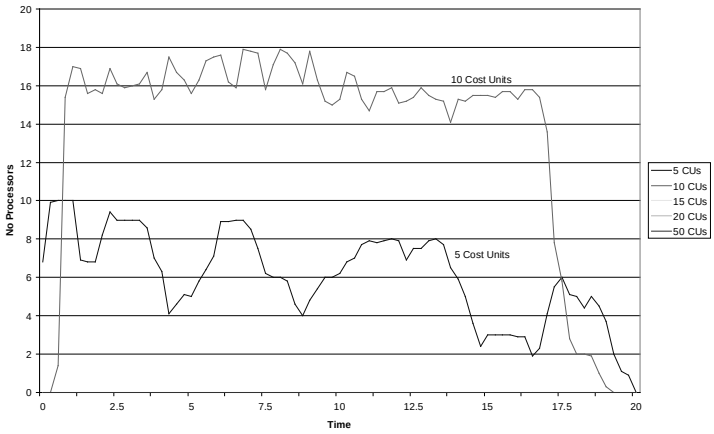
Graph 2 - GUSTO Usage for Ionization Chamber Study



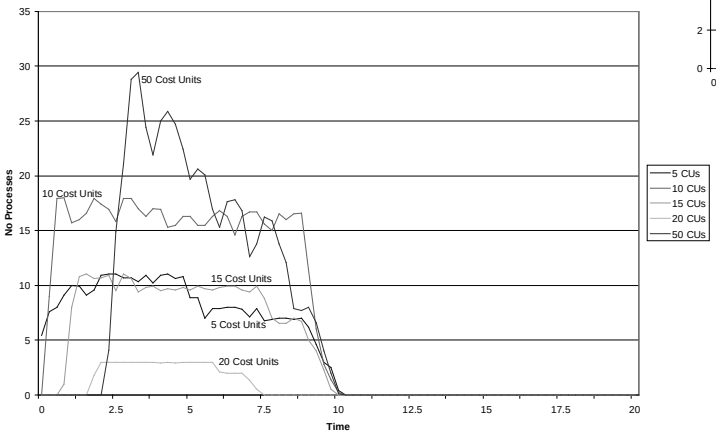
Graph 4 - GUSTO Usage for 15 Hour Deadline



Graph 3 - GUSTO Usage for 20 Hour Deadline



Graph 5 - GUSTO Usage for 10 Hour Deadline



The Grid Economy

RAJKUMAR BUYYA, DAVID ABRAMSON, AND SRIKUMAR VENUGOPAL

Invited Paper

This paper identifies challenges in managing resources in a Grid computing environment and proposes computational economy as a metaphor for effective management of resources and application scheduling. It identifies distributed resource management challenges and requirements of economy-based Grid systems, and discusses various representative economy-based systems, both historical and emerging, for cooperative and competitive trading of resources such as CPU cycles, storage, and network bandwidth. It presents an extensible, service-oriented Grid architecture driven by Grid economy and an approach for its realization by leveraging various existing Grid technologies. It also presents commodity and auction models for resource allocation. The use of commodity economy model for resource management and application scheduling in both computational and data grids is also presented.

Keywords*—Distributed computing, grid economy, resource management, utility computing.*

I. INTRODUCTION

Inspired by the electrical power Grid's pervasiveness, ease of use, and reliability, computer scientists in the mid-1990s began exploring the design and development of an analogous infrastructure called the *computational power Grid* [23] for wide-area parallel and distributed computing. The motivation for computational Grids was initially driven by large-scale, resource (computational and data) intensive scientific applications that require more resource than a single computer (PC, workstation, supercomputer, or cluster) could provide in a single administrative domain. A Grid enables the sharing, selection, and aggregation of a wide variety of geographically distributed resources including supercomputers, storage systems, data sources, and specialized devices owned by different organizations for solving large-scale resource intensive problems in science, engineering, and commerce.

Manuscript received March 1, 2004; revised June 1, 2004.

R. Buyya is with the Grid Computing and Distributed Systems Laboratory and NICTA Victoria Laboratory, Department of Computer Science and Software Engineering, University of Melbourne, VIC 3010, Australia (e-mail: raj@csse.unimelb.edu.au).

D. Abramson is with the School of Computer Science and Software Engineering, Monash University, VIC 3163, Australia.

S. Venugopal is with the Grid Computing and Distributed Systems Laboratory, Department of Computer Science and Software Engineering, University of Melbourne, VIC 3010, Australia.

Digital Object Identifier 10.1109/JPROC.2004.842784

To build a Grid, the development and deployment of a number of services is required. They include *low-level services* such as security, information, directory, and resource management [resource trading, resource allocation, and quality-of-services (QoSs)] and *high-level services/tools* for application development, resource management, and scheduling (resource discovery, access cost negotiation, resource selection, scheduling strategies, QoSs, and execution management) [23], [35], [52], [50], [51]. Among them, the two most challenging aspects of Grid computing are resource management and scheduling.

A. Resource Management and Scheduling

In Grid environments, the *producers* (resource owners) and *consumers* (resource users) have different goals, objectives, strategies, and supply-and-demand patterns. More importantly both resources and end users are geographically distributed in different time zones. The most commonly used approaches for managing such complex environments are driven by s *ystem-centric* and *user-centric* policies. System-centric is a traditional approach to resource management that attempts to optimize system-wide measure of performance and is commonly used in managing resources in single administrative domains. User-centric approaches, on the other hand, concentrate on delivering maximum utility to the users of the system based on their QoS requirements, i.e., a guarantee of certain levels of performance based on the attributes that the user finds important such as the deadline by which his jobs have to be completed. Enforcing QoS requires a system of rewards and penalties and, hence, it is common to find user-centric approaches driven by economic principles.

System-centric Grid resource management systems such as Legion [64], Condor [41], AppLeS PST [15], [21], NetSolve [20], PUNCH [47], and XtremWeb [17] adopt a *conventional strategy*, where a scheduling component decides which jobs are to be executed at which resource based on cost functions driven by system-centric parameters. They aim to enhance the system throughput, utilization, and complete execution at the earliest possible time rather than improving the utility of application processing. They do not take resource access cost (price) into consideration, which means that the value of processing applications at any time is treated the

0018-9219/$20.00 © 2005 IEEE

same, which is not the case in reality—the value should be higher when there is a production schedule deadline. The end user does not want to pay the highest price but wants to negotiate a particular price based on the demand, value, priority, and available budget.

In an economy-based approach, scheduling decisions are made dynamically at runtime and are driven and directed by the end-users requirements. While a conventional cost model often deals with software and hardware costs for running applications, an economy model primarily charges the end user for services that they consume based on the value they derive from it. Pricing policies are based on the demand from the users and the supply of resources is the main driver in the competitive, economic market model. Therefore, a user competes with other users and a resource owner with other resource owners.

Traditional approaches use centralized policies that need complete state information and a common fabric management policy, or a decentralized consensus based policy. The economic approach provides a fair basis in successfully managing the decentralization and heterogeneity that is present in human economies. Competitive economic models provide algorithms/policies and tools for resource sharing or allocation in Grid systems. These models can be based on bartering or prices. In the *bartering-based model*, all participants need to own resources and trade resources by exchanges (e.g., storage space for CPU time). In the *price-based model*, the resources have a price, based on the demand, supply, value, and the wealth in the economic system. In addition, it enhances the social structure of the Grid thereby ensuring its stability and efficiency [16].

The resource management and scheduling systems for Grid computing need to manage resources and application execution depending on resource consumers' and owners' requirements, and they need to continuously adapt to changes in the availability of resources. This requirement introduces a number of challenging issues that need to be addressed such as, site autonomy, heterogeneous substrate, policy extensibility, resource allocation or co-allocation, online control, resource trading, and QoS-based scheduling. A number of Grid systems (such as Globus [24] and [32]) have addressed many of these issues with the exception of *resource trading* and *quality of service-based scheduling.* The Grid Economy framework presented in this chapter addresses these two issues. It leverages existing middleware technologies and provides new services that are essential for resource trading and aggregation, depending on their availability, capability, cost, and users' QoS requirements.

B. Computational Economy: Assessing Wants and Needs

In an economic-based Grid computing environment, resource management systems need to provide mechanisms and tools that allow resource consumers (end users) and providers (resource owners) to express their requirements and facilitate the realization of their goals. Resource consumers need a utility model—how consumers demand resources and their preference parameters, and a broker that supports resource discovery and strategies for dynamically scheduling applications on distributed resources at runtime depending on their availability, capability, and cost along with user-defined QoS requirements. The resource providers need tools and mechanisms that support price specification and generation schemes to increase system utilization, and protocols that support service publication, trading, and accounting. For the market to be competitive and healthy, coordination mechanisms are required so that equilibrium is achieved, i.e., the supply of a service equals the quantity demanded.

Numerous economic models including microeconomic and macroeconomic principles for resource management have been proposed in the literature [3], [26], [40], [43], [51], [56], [61]. These include: commodity market models, posted price models, bargaining models, tendering, or contract-net models, auction models, bid-based proportional resource sharing models, cooperative bartering models, and monopoly and oligopoly.

In general, the benefits of Grid economies can be listed as follows.

- It helps in building a large-scale Grid as it offers incentive for resource owners to contribute their resources for others to use and profit from it.
- It helps in regulating the supply and demand for resources.
- It offers an economic incentive for users to reduce their priority in favor of incurring a lesser expense and, thus, encourages the solution of time critical problems first.
- It provides a common basis for comparing conflicting needs by allowing users to express their requirements and objectives in currency terms.
- It offers uniform treatment of all resources. That is, it allows trading of everything including computational power, memory, storage, network bandwidth/latency [58], data, and devices or instruments.
- It helps in building a highly scalable system as the decision-making process is distributed across all users and resource owners.
- It supports a simple and effective basis for offering differentiated services for different applications at different times.

Finally, it places the power in the hands of both resource owners and users—they can make their own decisions to maximize the utility gained and profit.

C. Requirements for Economic-Based Grid Systems

Economic-based resource management systems need to provide mechanisms and tools that allow resource consumers (end users) and providers (resource owners) to express their requirements and facilitate decision-making to further their objectives [9]. That is, they need 1) the means to express their valuations and objectives [*value expression*], 2) scheduling policies to translate them to resource allocations [*value translation*], and 3) mechanisms to enforce selection and allocation of differential services, and dynamic adaptation to changes in their availability at runtime [*value enforcement*]. Similar requirements are raised [7] for market-based systems in a single administrative domain environment such as clusters. However, they are limited to co-operative economic models since they aim for social welfare. Grids need to use

competitive economic models as resource providers and resource consumers have varying goals, strategies, and requirements that vary with time.

Essentially, resource consumers need a *utility model*—to allow them to specify resource requirements and constraints. They need *brokers* that provide strategies for choosing appropriate resources [*value translation*] and dynamically adapt to changes in resource availability at runtime to meet user requirements [*value enforcement*]. The resource owners need mechanisms for *price generation schemes* to increase system utilization and *protocols* that help them offer competitive services [*value expression*]. Grid resources have their schedulers (e.g., OS or queuing system) that allocate resources [*value translation*]. Some research systems support resource reservation in advance (e.g., reserving a slot from time t1 to t2 using the Globus GARA [25] and bind a job to it) and allocate resources during reserved time [*value enforcement*]. A number of research systems have explored QoS based resource (e.g., CPU time and network bandwidth [58], [3]) allocation in operating systems and queuing systems, but the inclusion of QoS into mainstream systems has been slow paced (e.g., the Internet mostly uses the best effort allocation policy [35], but this is changing with IPv6 [5]).

An economic approach to Grid computing introduces a number of new issues like resource trading and QoS-based scheduling in addition to those such as site autonomy, heterogeneous substrate, policy extensibility, online control already addressed by existing Grid systems. To address these new issues, the economy-based Grid systems need to support the following.

- An information and market directory for publicizing Grid entities.
- Models for establishing the value of resources.
- Resource pricing schemes and publishing mechanisms.
- Economic models and negotiation protocols.
- Mediators to act as a regulatory agency for establishing resource value, currency standards, and crisis handling.
- Accounting, Billing, and Payment Mechanisms.
- Users' QoS requirements-driven brokering/scheduling systems.

II. Representative Works

Various criteria used for judging effectiveness of a market model are [69]: social welfare (global good of all), Pareto efficiency (global perspective), individual rationality (better off by participating in negotiation), stability (mechanisms that cannot be manipulated, i.e., behave in the desired manner), computational efficiency (protocols should not consume too much computation time), and distribution and communication efficiency (communication overhead to capture a desirable global solution).

Several research systems (see Table 1) have explored the use of different economic models for trading resources to manage resources in different application domains: CPU cycles, storage space, database query processing, and distributed computing, They include Spawn [10], Popcorn [48], Java Market [72], Enhanced MOSIX [73], JaWS [66], Xenoservers [14], D'Agents [28], Rexec/Anemone [6], Mojo Nation [45], Mariposa [44], Mungi [18], Stanford Peers [8], G-Commerce [62], OCEAN [39], Nimrod-G [52], and GridSim [49], and Gridbus [57].

Each of the resource management systems presented in Table 1 follows a single model for resource trading. They have been designed with a specific goal in mind either for CPU or storage management. In order to use some of these systems, applications have to be designed using their proprietary programming models, which is generally discouraging, as applications need to be specifically developed for executing on those systems. Also, resource trading and job management modules have been developed using monolithic system architecture that limits their extensibility.

III. Grid Architecture for Computational Economy

A distributed grid architecture for computational economy (GRACE) is shown in Fig. 1 and has been enhanced to support computational, data, and service-oriented Grids. This architecture is generic enough to accommodate different economic models used for resource trading. The key components of the Grid include the following.

- Grid User with Applications (sequential,arametric, parallel, or collaborative applications).
- Programming Environments.
- User-Level Middleware and Tools such as GRBs.
- Core Grid Middleware (services for resource trading and coupling distributed wide area resources).
- Grid Service Providers (GSPs).

GRACE provides services that help both resource owners and users maximize their objective functions. The resource providers can contribute their resources to the Grid and charge for services. They can use GRACE mechanisms to define their charging and access policies and the GRACE resource trader works according to those policies. The users interact with the Grid by defining their requirements through high-level tools such as resource brokers. The resource brokers work for the consumers and attempt to maximize user utility. They can use GRACE services for resource trading and identifying GSPs that meets its requirements.

Both GRBs and GSPs can initiate resource trading and participate in the interaction depending on their requirements and objectives. GRBs may invite bids from a number of GSPs and select those that offer the lowest service costs and meet their deadline and budget requirements. Alternatively, GSPs may invite bids in an auction and offer services to the highest bidder as long as its objectives are met. Both GSPs and GRBs have their own utility functions that must be satisfied and maximized. The GRBs perform a cost-benefit analysis depending on the deadline (by which the results are required) and budget available (the amount of money the user is willing to invest for solving the problem). The resource owners decide their pricing based on various factors. They may charge different prices for different users for the same service or it can vary depending on the specific user demands. Resources may have different prices based on environmental influences such as the availability of larger core memory and better communication bandwidth with the outside world.

Grid brokers (note that in a Grid environment each user has his/her own broker as his agent) may have different goals

Table 1 Computational Economy Based Distributed Resource Management Systems

System Name	Economic Model	Platform	Remarks
Mariposa [44] (UC Berkeley)	Bidding (Tendering/ ContractNet). Pricing based on load and historical info.	Distributed database.	It supports budget-based query processing and storage management.
Mungi [18] (University of New South Wales)	Commodity market (renting storage space that increases as available storage runs low, forcing users to release unneeded storage.)	Storage servers.	It supports storage objects based on bank accounts from which rent is collected for the storage occupied by objects.
Popcorn [48] (Hebrew University)	Auction. (Highest bidder gets access to resource and it transfers credits from buyer to the seller account.)	Web browsers. (*Popcorn parallel code* runs within a browser of CPU cycles seller.)	Popcorn API-based parallel applications need to specify a budget for processing each of its modules.
Java Market [72] (Johns Hopkins University)	QoS based computational market. (The resource owner receives f(j, t) award for completing f in time t.)	Web browsers. (JavaMarket runs *standard Java Applets* within a browser).	One can sell CPU cycles by pointing Java-enabled browser to Portal & allow execution of Applets.
Enhanced MOSIX [73] (Hebrew U., Israel)	Commodity market (resource cost of each node is known)	Clusters of computers (Linux PCs)	It supports process migration such that overall cost of job execution is kept low.
JaWS [66] (University of Crete)	Bidding (Tendering)	Web browsers	It is similar to Popcorn.
Xenoservers [14] (University of Cambridge)	Bidding (Proportional resource sharing)	Single computer	Accounted execution of un-trusted code.
D'Agents [28] (Dartmouth College)	Bidding (Proportional resource sharing)	Single computer or Mobile Agents	Agents bid function is proportional to benefit.
Rexec/Anemone [6] (UC Berkeley)	Bidding/Auction (for proportional resource sharing)	Clusters (A market-based Cluster Batch Queue System)	Users assign utility value to their application and system allocates resources proportionally.
Mojo Nation [45] (Autonomous Zone Industries, CA)	A Credit-based partnership and/or bartering model. (Contributors earn credits by sharing storage and spend them when required)	Network storage.	It is a content-sharing community network. It combines marketplace and bartering approach for file/resource sharing.
Spawn [10] (Xerox PARC)	Second-price/Vickery auction (uses sponsorship model for funding money to each task depending on some requirements)	Network on workstations. Each WS executes a single task per time slice	It supports execution of concurrent program expressed in the form of hierarchy of processes that expand and shrink size depending on the resource cost.
CSAR Supercomputing center [29] (University of Manchester)	Commodity market and priority-based model (they charge for CPU, memory, storage, and human support services)	MPPs, Crays, and Clusters, and Storage servers.	Any application can use this service and QoS is proportional to user priority and scheduling mechanisms.
Nimrod-G [52] (Monash University)	It supports economy models such as commodity market, spot market, and contract-net for price establishment.	World Wide Grid (resources Grid enabled using Globus middleware)	It is a *real* system that supports deadline and budget constrained algorithms for scheduling task-farming and data parallel applications on world-wide distributed resources depending on their cost, power, availability and users quality of service requirements.
GridSim [49] (Monash University)	Currently, it supports economic models similar to those used in Nimrod-G, but limited to them.	A Java-based discrete event toolkit for simulating Grid resources, users, applications, and brokers.	The economic Grid resource broker supports deadline and budget based time, cost, cost-time, and conservative time optimisation scheduling algorithms.
G-Commerce [62] (U. of California Santa Barbara)	Commodity and auctions	Simulates hypothetical consumers and produces.	It is exploring strategies for pricing Grid resources to enable resource trading.
OCEAN [39] (U. of Florida)	Continuous double auction	A Java based platform with distributed PCs.	It is exploring the use of continuous double auction for trading computational resources – in development.
Stanford Peers [8] (Stanford University)	Auctions with cooperative bartering in a cooperative sharing environment.	Simulates storage trading for content replication and archiving.	It demonstrates distributed resource trading policies based on auctions by *simulation* – in development.
Gridbus [57] (U. of Melbourne)	Both commodity and auction models for clusters, computational and data grids	Clusters and the World-Wide Grid	It provides services for ensuring end-to-end Quality of Services at various levels for users in cluster and Grid environments.

(e.g., different deadlines and budgets), and each broker tries to maximize its own good without concern for the global good. This needs to be taken into consideration in building automated negotiation infrastructure. In a *cooperative distributed computing or problem-solving environment* (like cluster computers or a federation of clusters), the system

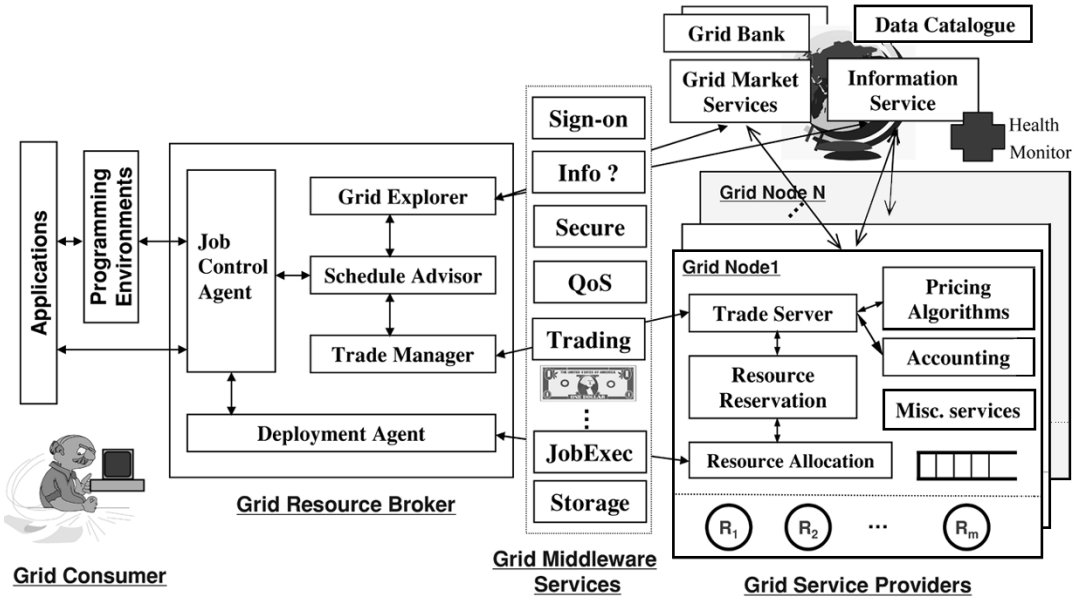


Fig. 1. A Generic Grid architecture for computational economy.

designers impose an *interaction protocol* (possible actions to take at different points) and a *strategy* (a mapping from one state to another and a way to use the protocol). This model aims for global efficiency as nodes cooperate toward a common goal. On the other hand, in Grid systems, brokers, and GSPs are provided with an interaction protocol, but they choose their own private strategy (similar to multi-agent systems), which cannot be imposed from outside. Therefore, the negotiation protocols need to be designed assuming a *noncooperative, strategic* perspective. In this case, the main concern is the social outcomes that follow given a protocol and that guarantees that each broker/GSPs desired local strategy is best for that broker/GSP and hence the broker/GSP will use it.

A. Grid Resource Broker (GRB)

The resource broker acts as a mediator between the user and Grid resources using middleware services. It is responsible for resource discovery, resource selection, binding of software, data, and hardware resources, initiating computations, adapting to the changes in Grid resources and presenting the Grid to the user as a single, unified resource. The resource broker consists of the following components.

- *Job Control Agent (JCA):* This is a persistent control engine responsible for shepherding a job through the system. It coordinates with schedule adviser for schedule generation, handles actual creation of jobs, maintenance of job status, interacting with clients/users, schedule advisor, and dispatcher.
- *Schedule Advisor (Scheduler):* This is responsible for resource discovery (using the Grid explorer), resource selection and job assignment (schedule generation) to ensure that the user requirements are met.
- *Grid Explorer (GE):* This is responsible for resource discovery by interacting with the Grid-information server and identifying the list of authorized machines, and keeping track of resource status information.
- *Trade Manager (TM):* This works under the direction of resource selection algorithm (the schedule advisor) to identify resource access costs. It uses market directory services and GRACE negotiation services for trading with Grid service providers (i.e., their representative trade servers).
- *Deployment Agent (DA):* It is responsible for activating task execution on the selected resource as per the scheduler's instruction and periodically updates the status of task execution to JCA.

B. Core Middleware Level and Grid Economy

Traditionally core Grid middleware focused on providing services required for secure and uniform access to distributed resources. They include security, single sign-on, remote process management, storage access, data management, and information services. These services are being standardized via efforts such as Web Services Resource Framework (WSRF) [33] to support the use of service-oriented architectures in distributed systems and applications development. Core middleware technologies such as Globus are aiming at providing standards-based software services. In the Grid economy context, GSPs specifically need to deal with the following components that can be considered as part of the services-driven next-generation core grid middleware.

- *Grid Market Directory (GMD):* It allows resource owners to publish their services in order to attract consumers.
- *Grid Trade Server (GTS):* This is a resource owner agent that negotiates with resource users and sells access to resources. It aims to maximize the resource utility and profit for its owner. It consults pricing policies during negotiation and directs the accounting system to record resource consumption and to bill the user according to the agreed pricing policy.
- *Pricing Policies:* These define the prices that resource owners would like to charge users. The resource

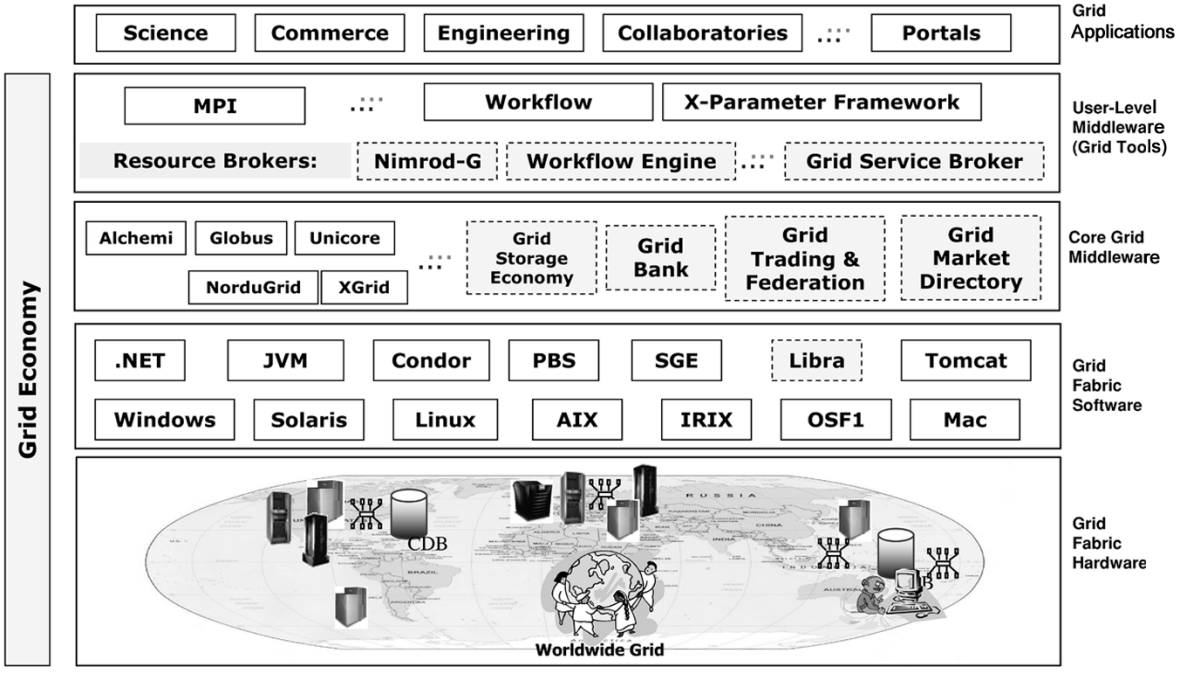


Fig. 2. Scenario for realizing the GRACE framework.

owners may follow various policies to maximize their profit and resource utilization and the price they charge may vary with time and user population. Also, the pricing can be driven by demand and supply as in the real market environment. That is, within commodity market model, pricing is essentially determined by objective functions of service providers and users. The pricing policy can also be based on auction. Within auction based economic model, pricing is driven by how much value users place on the service and access to Grid services is won by the bidder whose valuation comes closest to that of the resource owner.

- *Resource Accounting and Charging:* components such as Grid Bank [1] along with QBank [65] are responsible for recording resource usage and bill the user as per the usage agreement between resource broker (a user agent) and trade server (resource owner agent).

The service providers publish their services through the GMD. They use Grid trading services' declarative language for defining cost specification and their objectives such as access price for various users for different times and durations, along with possibilities of offering discounts to attract users during off-peak hours. The GTS can employ different economic models in providing services. The simplest would be a commodity model wherein the resource owners define pricing strategies including those driven by the demand and resource availability. The GTS can act as auctioneer if the Auction-based model is used in deciding the service access price or an external auctioneer service can be used.

A layered architecture for the realization of the GRACE framework is shown in Fig. 2. It offers Grid economy infrastructure that co-exists with or is built on top of the existing middleware such as Globus, Alchemi [4] and Unicore [27]. The impact of Grid economy can be felt at various levels of system architecture: local resource management, resource access mediator services provided by core middleware, resource brokers while making selection of resources, a programming framework and policy that allocates budget for different activities of an application.

- Problem solving environments with built in schedulers (e.g., ActiveSheets [12] on Nimrod-G [11]).
- Programming frameworks and development tools (e.g., Nimrod parameter specification language [13]).
- A resource broker (e.g., Nimrod-G).
- Various resource trading protocols.
- A mediator for negotiating between users and Grid service providers (Grid Market Directory [31]).
- A deal template for specifying resource requirements and services offers.
- A trade server.
- A pricing policy specification.
- Accounting (e.g., QBank [65]) and payment management (GridBank [1]).
- A local resource management system allocating resources based on service-level agreements (e.g., Libra [30]).

The new middleware services being proposed are designed to offer low-level services that co-exist with existing low-level middleware services and infrastructure. Higher-level services and tools such as the Nimrod-G Resource Broker, which uses economic models suitable for meeting the user requirements, can use these core services.

IV. Economy Within Computational Grids

The integration of computational economy as part of a scheduling system greatly influences the way computational resources are selected to meet the user requirements. The users should be able to submit their application along with their requirements to a scheduling system such as Nimrod-G,

which can process the application on the Grid on the user's behalf and try to complete the assigned work within a given deadline and cost. The deadline represents a time by which the user requires the result, and is often imposed by external factors like production schedules or research deadlines.

To arrive at a scheduling decision, the scheduling system needs to take various parameters into consideration including resource architecture and configuration, resource state (such as CPU load, memory available, disk storage free), resource requirements of an application, resource availability, network bandwidth, Load, and Latency, historical Information such job consumption rate. The important parameters of computational economy that can influence the way resource scheduling is done are the following.

- Resource Cost (set by its owner).
- Price (that the user is willing to pay).
- Deadline (the period by which an application execution needs to be completed).

The scheduler can use the information gathered by a resource discoverer and also negotiate with resource owners to establish service price. The resource that offers the best price and meets resource requirements can eventually be selected. This can be achieved by resource reservation and bidding. If the user deadline is relaxed, the chances of obtaining low-cost access to resources are high. The cost of resources can vary with time and the resource owner will have the full control over deciding access cost. Further, the cost can vary from one user to another. The scheduler can even solicit bids from resource providers in an open market, and select the feasible service-provider(s). To accomplish this, we need scheduling algorithms that take the application processing requirements, Grid resource dynamics, and the user QoS requirements such as the deadline, budget, and their optimization preference into consideration. In this section, we discuss deadline and budget constrained (DBC) algorithms that we developed for scheduling parameter sweep applications on globally distributed Grid resources.

A. Scheduling Algorithms

The parameter sweep applications, created using a combination of task and data parallel models, contain a large number of independent jobs operating different data sets. A range of scenarios and parameters to be explored are applied to the program input values to generate different data sets. The programming and execution model of such applications resemble the SPMD (Single Program Multiple Data) model. The execution model essentially involves processing N independent jobs (each with the same task specification, but a different dataset) on M distributed computers where N is, typically, much larger than M.

The scheduling and orchestration of the execution of parameter sweep applications on world-wide distributed computers appears simple, but complexity arises when users place QoS constraints like deadline (execution completion time) and computation cost (budget) limitations. Such a guarantee of service is hard to provide in a Grid environment since its resources are shared, heterogeneous, distributed in nature, and owned by different organizations having their own policies and charging mechanisms. In addition, scheduling algorithms need to adapt to the changing load and resource availability conditions in the Grid in order to achieve performance and at the same time meet the deadline and budget constraints. In our Nimrod-G application level resource broker (also called an application level scheduler) for the Grid, we have incorporated four adaptive algorithms for deadline and budget constrained scheduling [56].

- Cost Optimization, within time and budget constraints.
- Time Optimization, within time and budget constraints.
- Conservative Time Optimization, within time and budget constraints.
- Cost-Time Optimization, within time and budget constraints.

Table 2 Deadline and Budget Constrained Scheduling Algorithms and Objectives

Scheduling Algorithm Strategies	*Execution Time (not beyond the deadline)*	*Execution Cost (not beyond the budget)*
Cost Optimisation	Limited by deadline	Minimise
Time Optimisation	Minimise	Limited by budget
Conservative Time Optimisation	Limited by deadline	Limited by budget
Cost-Time Optimisation	Limited by deadline	Limited by budget

The role of deadline and budget constraints in scheduling and objectives of different scheduling algorithms are illustrated in Table 2. In this following section we present two cost and time optimization-based scheduling algorithms and their performance results.

The *Time Optimization scheduling* algorithm attempts to complete the experiment as quickly as possible, within the budget available. A description of the core of the algorithm is as follows.

1) For each resource, calculate the next completion time for an assigned job, taking into account previously assigned jobs and job consumption rate.
2) Sort resources by next completion time.
3) Assign one job to the first resource for which the cost per job is less than or equal to the remaining budget per job.
4) Repeat the above steps until all jobs are assigned.

The *Cost Optimization scheduling* algorithm attempts to complete the experiment as economically as possible within the deadline.

1) Sort resources by increasing cost.
2) For each resource in order, assign as many jobs as possible to the resource, without exceeding the deadline.
3) Note that the implementations of all the above algorithms contain extra steps for dealing with the initial start-up (when the average completion times are unknown), and for when all jobs cannot be assigned to resources (infeasible schedules).

B. Scheduling Experiments

We have performed a number of deadline and budget constrained scheduling experiments with different requirements at different times by selecting different sets of resources available in the World Wide Grid (WWG) [55] testbed

```
#Parameters Declaration
parameter angle_degree integer range from 1 to 165 step 1;
parameter time_base_value integer default 5;

#Task Definition
task main
   #Copy necessary executables depending on node type
   copy calc.$OS node:calc
   #Execute program with parameter values on remote node
   node:execute ./calc $angle_degree $time_base_value
   #Copy results file to use home node with jobname as extension
   copy node:output ./output.$jobname
endtask
```

Fig. 3. Nimrod-G parameter sweep processing specification.

Table 3 The WWG Tested Resources Used in Scheduling Experiments, Job Execution and Costing

Resource Type & Size (No. of nodes)	Organization & Location	Grid Services and Fabric	Price (G$ per CPU sec.)	Jobs Executed on Resources	
				Time_Opt	Cost_Opt
Linux cluster (60 nodes)	Monash, Australia	Globus, GTS, Condor	2	64	153
Solaris (Ultra-2)	Tokyo Institute of Technology, Japan.	Globus, GTS, Fork	3	9	1
Linux PC (Prosecco)	CNUCE, Pisa, Italy	Globus, GTS, Fork	3	7	1
Linux PC (Barbera)	CNUCE, Pisa, Italy	Globus, GTS, Fork	4	6	1
Sun (8 nodes)	ANL, Chicago, USA	Globus, GTS, Fork	7	42	4
SGI (10 nodes)	ISI, Los Angeles, USA	Globus, GTS, Fork	8	37	5
			Total Experiment Cost (G$)	237000	115200
			Time to Complete Experiment (Min.)	70	119

during each experiment. They can be categorised into the following scenarios:

- cost optimization scheduling during australian peak and off-peak times,
- cost and time optimization scheduling using cheap local and expensive remote resources;
- large scale scheduling using cost and time optimization algorithms.

We briefly discuss the WWG testbed followed by a detailed discussion on these scheduling experiments.

1) The World-Wide Grid (WWG) Testbed: To enable our empirical research and experimentations in distributed computational economy and Grid computing, we created and expanded a testbed called the World-Wide Grid (WWG) in collaboration with colleagues from numerous organizations around the globe. The contributing organizations and the WWG resources are located in five continents: Asia, Australia, Europe, North America, and South America.

The WWG testbed contains numerous computers with different architecture, capability, and configuration. They include PCs, workstations, SMPs, clusters, and vector supercomputers running operating systems such as Linux, Sun Solaris, IBM AIX, SGI IRIX, and Compaq Tru64. Further, the systems use a variety of job management systems such as OS-Fork, NQS, Condor, RMS, PBS, and LSF. These system characteristics can be identified by accessing the GIS (Grid Information Service) provided by middleware systems such as Globus running on each resource.

2) Parameter Sweep Application: We have created a hypothetical parameter sweep application (PSA) that executes a CPU intensive program with 165 different parameter scenarios or values. The program *calc* takes two input parameters and saves results into a file named "output." The first input parameter *angle_degree* represents the value of angle in degree for processing trigonometric functions. The program *calc* needs to be explored for angular values from 1 to 165 degrees. The second parameter time_base_value indicates the expected calculation complexity in minutes plus 0–60 s positive deviation. That means the program *calc* is expected to run for anywhere between 5 and 6 min on resources with some variation depending on resource capability. A plan

file modeling this application as a parameter sweep application using the Nimrod-G parameter specification language is shown in Fig. 3. The first part defines parameters and the second part defines the task that needs to be performed for each job. As the parameter *angle_degree* is defined as a range parameter type with values varying from 1 to 165 in step of 1, it leads to the creation of 165 jobs with 165 different input parameter values. To execute each job on a Grid resource, the Nimrod-G resource broker, depending on its scheduling strategy, first copies the program executable(s) and necessary data to a Grid node, then executes the program, and finally copies results back to the user home node and stores output with job number as file extension.

3) Cost and Time Optimization Scheduling Using Local and Remote Resources: This experiment demonstrates the use of cheap local resources and expensive remote resources together for processing a parameter sweep application (same as used in the previous scheduling experiment) containing 165 CPU-intensive jobs, each running approximately 5-min duration. We have set the deadline of 2 h (120 min) and budget of 396 000 (G$ or tokens) and conducted experiments for two different optimization strategies.

- *Optimize for Time*—this strategy produces results as early as possible, but before a deadline and within a budget limit.
- *Optimize for Cost*—this strategy produces results by deadline, but reduces cost within a budget limit.

In these scheduling experiments, the Nimrod-G resource broker employed the *commodity market* model for establishing a service access price. The broker established connection with the Grid Trader running on resource providers' machines to obtain service prices at runtime. The broker architecture is generic enough to use any of the protocols discussed in [54] for negotiating access to resources and choosing appropriate ones. The access price varies for local and remote users: users are encouraged to use local resources since they are available at cheaper price. Depending on the deadline and the specified budget, the broker develops a plan for assigning jobs to resources. While doing so it does dynamic load profiling to establish the user job consumption rate for each resource. The broker uses this information to adapt itself to the changing resource conditions including failure of resources or jobs on the resource.

We have used a subset of resources of the WWG testbed in these scheduling experiments. Table 3 shows resources details such as architecture, location, and access price along with type of Grid middleware systems used in making them Grid enabled. These are shared resources and hence they were not fully available to us. The access price indicated in the table is being established dynamically using the GRACE resource trading protocols (commodity market model). The access price are artificial, however, they assigned to reflect the offering of differentiated services at different costs as in the real-world marketplace.

The number of jobs in execution on resources (Y-axis) at different times (X-axis) during the experimentation is shown in Fig. 4(a) and 4(b) for the time and cost optimization scheduling strategies, respectively. In the first (time minimization) experiment, the broker selected resources in such a way that the whole application execution is completed at the earliest

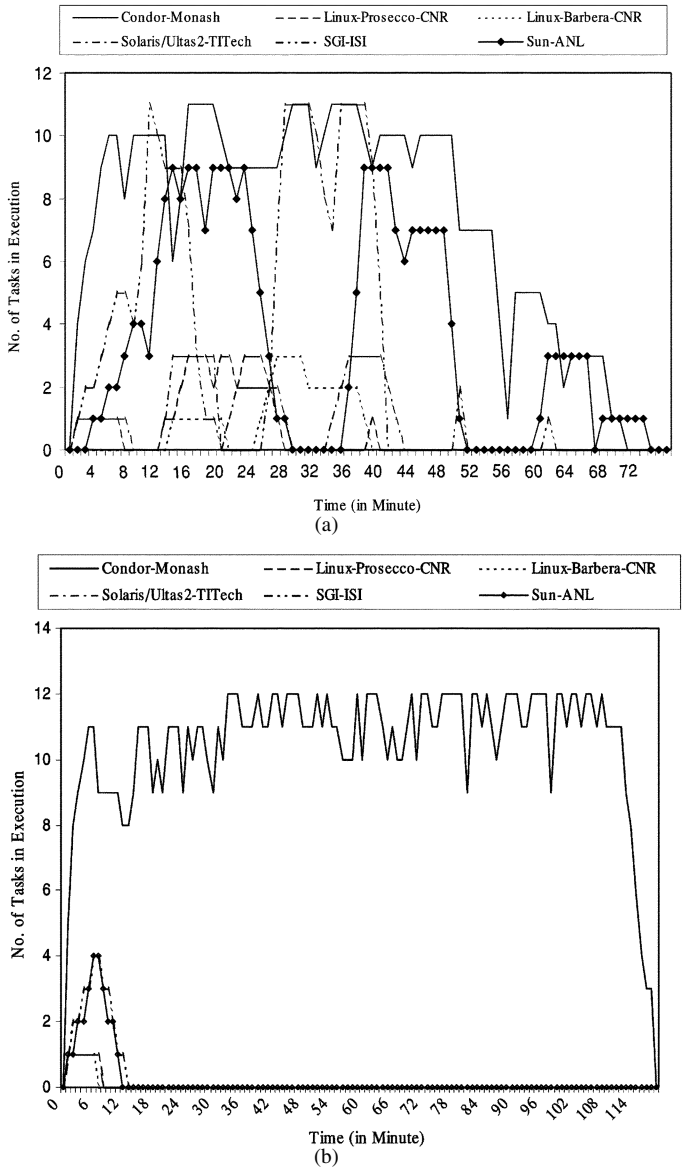


Fig. 4. (a) Resource selection in deadline and budget constrained time optimization scheduling. (b) Resource selection in deadline and budget constrained cost optimization scheduling.

time for a given budget. In this experiment, it completed execution of all jobs within 70 min and spent 237 000 G$. In the second experiment (cost minimization), the broker selected cheap resources as much as possible to minimize the execution cost whilst still trying to meet the deadline (completed in 119 min) and spent 115 200 G$. After the initial calibration phase, the jobs were distributed to the cheapest machines for the remainder of the experiment. The processing expense of the time-optimization scheduling experiment is much larger than the cost-optimization scheduling experiment due to the use of expensive resources to complete the experiment early. The results show that our Grid brokering system can take advantage of economic models and user input parameters to meet their requirements.

V. Economy Within Data Grids

A data-intensive computing environment [60] can be perceived as a real-world economic system wherein there are

Table 4 Resources Within Belle Testbed Used for Evaluation, Their Roles and Costing.

Organization	Machine details	Role	Cost (in G$ per Cpusec)	Total Jobs Executed	
				Time	Cost
Dept. of Computer Science, University of Melbourne.	*belle.cs.mu.oz.au* IBM eServer, 4 CPU, 2GB RAM, 70 GB HD, Linux	Broker host, Data host, NWS server	N.A. (Not used as a compute resource)	--	--
School of Physics, University of Melbourne	*fleagle.ph.unimelb.edu.au* PC, 1 CPU, 512 MB RAM, 70 GB HD, Linux	Replica Catalog host, Data host, Compute resource, NWS sensor	2	3	94
Dept. of Computer Science, University of Adelaide	*belle.cs.adelaide.edu.au* IBM eServer, 4 CPU (only 1 available) , 2GB RAM, 70 GB HD, Linux	Data host, NWS sensor	N.A. (Not used as a compute resource)	--	--
Australian National University, Canberra	*belle.anu.edu.au* IBM eServer, 4 CPU, 2GB RAM, 70 GB HD, Linux	Data host, Compute resource, NWS sensor	4	2	2
Dept of Physics, University of Sydney	*belle.physics.usyd.edu.au* IBM eServer, 4 CPU (only 1 available), 2GB RAM, 70 GB HD, Linux	Data host, Compute resource, NWS sensor	4	72	2
Victorian Partnership for Advanced Computing, Melbourne	*brecca-2.vpac.org* 180 node cluster (only head node used), Linux	Compute resource, NWS sensor	6	23	2

producers and consumers of data distributed geographically across multiple organizations. Producers are entities which generate the data and control its distribution via mirroring at various replica locations around the globe. They lay down policies for replication that are guided by various criteria such as minimum bandwidth, storage and computational requirements, data security and access restrictions and data locality issues. However, information about the data replicas is assumed to be available through a data catalogue mechanism such as the Globus Replica Catalog [67]. An example of such a system would be the tier-level model proposed by the MONARC [46] group within CERN for replicating the data produced by the Large Hadron Collider (LHC) [36], [71] for use within the ATLAS and CMS collaborations. The consumers in this system would be the users or, by proxy, their applications which need to analyze this data to produce meaningful results. The users may want to investigate specific datasets out of a set of hundreds and thousands and may have specific application requirements that need not be fulfilled at every computational site.

While scheduling of bandwidth or storage-intensive applications is currently a hot topic in the Grid research community [34], [38], most of them give concentrate on the job turnaround time and give less importance to the limitations of the network and storage resources such as bandwidth and storage caps or increasing cost with increasing usage. Also, in large collaborative environments such as scientific Grids, the presence of a large number of users can put a lot of pressure on the data infrastructure (i.e., network and storage elements). The pressure becomes more acute when a nontrivial percentage of the users are interested in the same datasets at the same time, thus causing heavy load on the servers on which the required datasets and its replicas are hosted. This denies service to not only the requestors of the datasets in question but also to those who require other datasets that are stored on the same servers. Such an effect is commonly observed in the Internet and the World Wide Web and a popular term, "Slashdot effect," is associated with it [59].

While a robust and adaptive replication mechanism can alleviate some of the above problems, the same problems of data access and transfer costs affect the effectiveness and efficiency of such a mechanism. Previous work in economy on data grids has tried to provide solutions to this problem through economy-driven data replication mechanisms [70]. Some cost models have been proposed for such economy-based replication [22]. However, no study has been made so far on the economic aspects of data processing by scheduling analysis jobs on various sites with varying execution, transfer, and storage costs.

In this section, we extend the notion of user-driven economic scheduling within computational grids to data grids. As in economy-based computational scheduling presented in Section IV above, the user supplies the deadline by which he wants his data analysis to be completed and his budget for the analysis job. He also supplies the minimization (cost or time) he wants to apply for scheduling the jobs. As is with the case in the sections above, these inputs are directly motivated by the priority and the urgency with which the user perceives his analysis task. Market forces guide this system to a stable state as users with less urgency will tend to save on their expenses by specifying longer deadlines.

A. Scheduling Algorithms

We have extended the previous work on economy-based scheduling and resource allocation within computational grids described above to data grids by implementing the cost and time minimization algorithms for distributed data-oriented applications. We define the cost to be minimized to be the sum of the processing cost, the data transfer (network) cost and the storage cost. Likewise, the time to be minimized is the sum of the job completion time and the data transfer time. The heuristic that we have implemented is a variation

of the Min-Min heuristic described in [42] and is detailed below:

1) Repeat for every scheduling interval while there exists unprocessed jobs
2) For every job, find the data file(s) that it is dependent on and locate the data hosts for those files.
3) Find a data-compute set (a set consisting of one compute resource for the execution and one data host for each file involved) that guarantees the *minimum cost* for that job.
4) Sort the jobs in the order of increasing cost.
5) Assign jobs from the sorted list starting with the *least expensive* job until either all the jobs are done or all the compute resources have been allocated their maximum *job limit*.

The *job limit* of each compute resource is calculated each scheduling interval on the basis of its present and past job completion ratio, i.e., the ratio of the number of jobs completed to the total number of jobs allocated but not completed since the previous scheduling interval. While the above listing shows only cost minimization, the same heuristic was followed in the case of time minimization except that the criterion in the second step was changed to the *minimum execution time required*.

B. Scheduling Experiments

To evaluate the scheduling heuristics, we have used an experimental setup modified from the one described in [68]. The testbed used in our experiments is detailed in Table 4. The Grid Service Broker [68], developed as part of the Gridbus Project [57], was extended to consider the price of transferring data over network links between the compute resources and the data hosts while scheduling jobs. In our experiments, although we have artificially assigned data transmission costs shown in Table 5, they can be linked to real costs as prescribed by ISPs (Internet Service Providers). We have used NWS (Network Weather Service) [63] for measuring the network bandwidths between the computational and the data sites. A number of the data hosts were also functioning as compute resources. The bandwidth between a data host and a compute resource on the same site was set to an arbitrarily high value (10000 Mbps) within the broker and the cost of network transfer in this case is set to zero.

We have also extended the synthetic parameter sweep application *calc* used for evaluating deadline and budget algorithms for computational grids in Section IV.A to be used on data grids. The extension implements transfer and processing of large data files that are located through querying the replica catalog as described in [68]. There are 100 data files, each 30 MB in size. Each of the five data hosts in Table 5 holds an equal number of these files and there is no replication of these files. Each job depends on exactly one of the input data files, thus creating 100 jobs. Since the output files are small (in KB) and are transferred to the broker upon completion of the jobs, we have not considered storage costs within this evaluation.

The experiments were carried out on 9th August 2004 between 6:00 p.m. and 10:00 p.m. AEST. Table 6 shows the summary of the results that were obtained. As is expected, cost minimization scheduling produces minimum computation and data transfer expenses whereas time minimization completes the experiments in the least time. The graphs in Fig. 5(a) and 5(b) show the number of jobs completed against time for the two scheduling strategies for data grids. It can be seen that these mirror the trends within the graphs in Fig. 4(a) and 4(b) for computational grids, i.e., time minimization used the more expensive but faster resources to execute jobs whereas cost minimization used the cheaper resource most to ensure a lower overall expense.

Tables 8 and 7 show the number of jobs that were allocated to a unique pair of a data host and a compute resource—the number of jobs which were executed on the compute node within the pair and accessed data from the corresponding data node—for time minimization and cost minimization, respectively. It can be seen that within time minimization, the jobs were allocated to the *best available* compute resource *nearest* (highest bandwidth available) to the source of data. For example, most of the jobs dealing with the data from ANU were executed at the University of Sydney node and those dealing with data from University of Melbourne Computer Science

Table 5 Network Costs Between the Data Hosts and the Compute Resources (in G$ Per MB)

Data Node / Compute Node	ANU	UniMelb Physics	Sydney Physics	VPAC
ANU	0	34.0	31.0	38.0
Adelaide CS	34.0	36.0	31.0	33.0
UniMelb Physics	40.0	0	32.0	39.0
UniMelb CS	36.0	30.0	33.0	37.0
Sydney Physics	35.0	33.0	0	37.0

Table 6 Summary of Evaluation Results

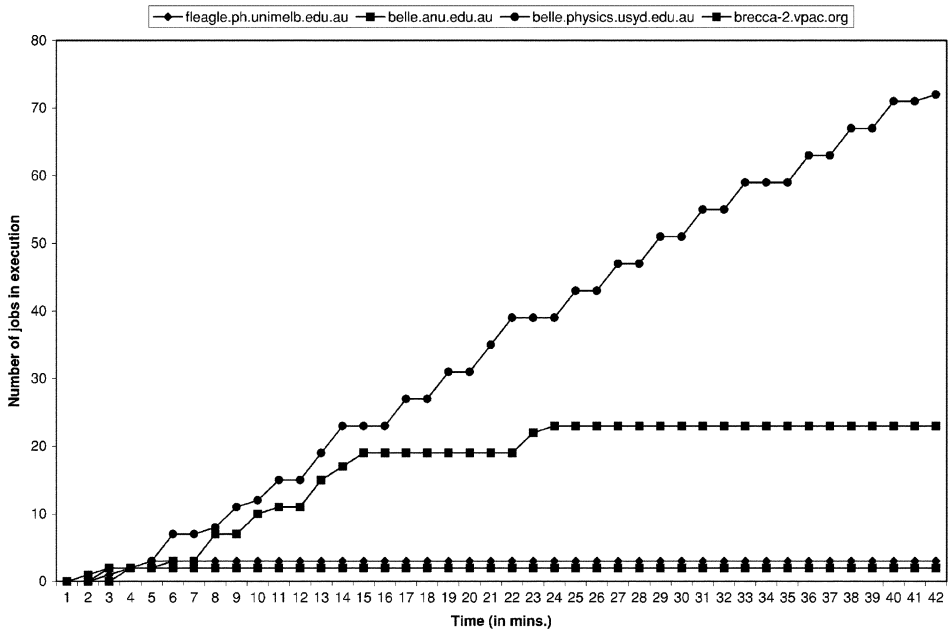
Scheduling strategy	Total Time Taken (mins.)	Compute Cost (G$)	Data Cost (G$)	Total Cost (G$)
Cost Minimization	71.07	26865	7560	34425
Time Minimization	48.5	50938	7452	58390

Table 7 Data Compute Allocation Matrix for Cost Minimization Scheduling

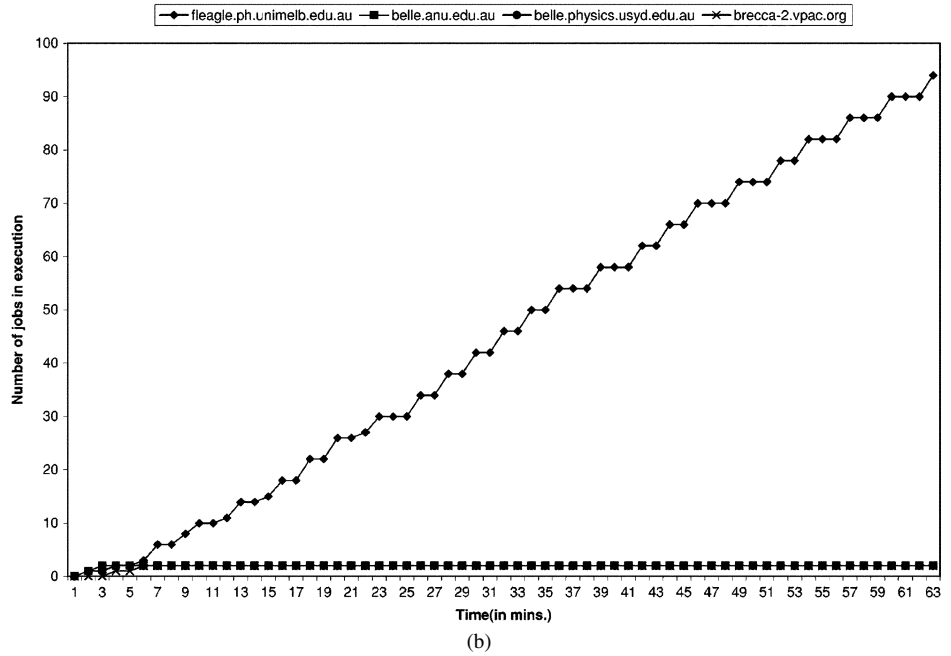
Compute Node / Data Node	UoM Phy	ANU	Sydney	VPAC
ANU	18	2	0	0
Adelaide	18	0	0	2
UoM CS	20	0	0	0
UoM Phy	20	0	0	0
Sydney	18	0	2	0

Table 8 Data Compute Allocation Matrix for Time Minimization Scheduling

Compute Node / Data Node	UoM Phy	ANU	Sydney	VPAC
ANU	0	2	18	0
Adelaide	0	0	19	1
UoM CS	0	0	2	18
UoM Phy	3	0	13	4
Sydney	0	0	20	0



(a)



(b)

Fig. 5. (a) Cumulative number of jobs completed versus time for time minimization scheduling in data grids. (b) Cumulative number of jobs completed versus time for cost minimization scheduling in data grids.

host were executed at the VPAC compute resource. Within cost minimization however, most of the jobs are allocated to the cheapest node without regard for the bandwidth cost. This is because the cost of processing dominated over the cost of transferring data, as seen from Table 6.

VI. Auction Models for Grid Resource Allocation

Auctioning has long been an important aspect of many economies. It provides a fair trading environment that has withstood the test of time. Auctions come in many different shapes and sizes, each born out of necessity or cultural differences. Despite these differences all auctions have common ideas and a basic principle that remains unchanged. Generally, every auction will consist of three entities: the seller, the bidders and the auctioneer. The auctioneer is responsible for the overall management of the auction.

In a Grid computing environment we can find entities which directly correspond to the three described in a traditional auction model. The seller is the GSP wishing to gain maximum profit out of its resources; the bidders are the Grid Resource Broker (GRB) which bid for access to resources keeping in mind the user requirements and constraints and the arbiter is Grid Market Auctioneer (GMA) which manages the auction process.

From this we can see how the auction model can be applied to a Grid computing environment. One important difference here is the correlation between assets in a traditional

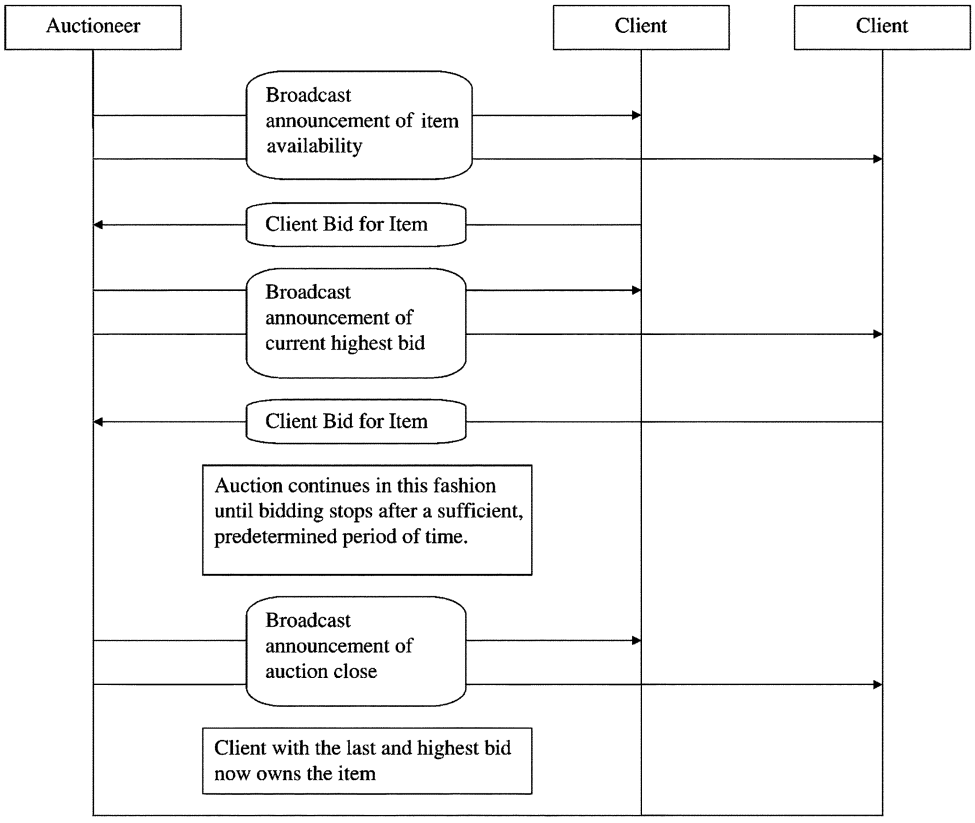


Fig. 6. The English Auction Interaction Model.

auction and resources in a Grid computing environment. In a traditional auction, the auctioneer handles all transactions whereas in a Grid resource auction, the bidders are bidding for the right to use a resource. So, once the auction is over, the auctioneer does not continue to mediate between buyer and seller. This is illustrated in the diagrams and is an important difference in how auctioning is applied to grids.

We have chosen to investigate English, Dutch and Double Auctions within Grid context.

A. English Auction

The English auction model is one of the most widely used auction models. It is an open auction where bidders can raise their bids in an attempt to outbid other prospective buyers. Fig. 6 outlines the basic interactions of the English auction.

The English auction has several positive and negative aspects. The fact that this is an open auction allows people to see what others are bidding. This helps clients to bid sensibly as well as giving you the opportunity to adjust your bid accordingly. Having many bidders is another benefit to the service providers as it allows them to sell for a much more competitive price.

One of the disadvantages of this system is the high level of communication required in this model. After each client makes a bid, the auctioneer must announce the new price to allow others to outbid. In a traditional auction this is not a problem as auctions are usually held with all bidders present, but it causes a serious problem when interested parties are on opposite sides of the world (as the case may be in a Grid computing environment). The problem of resources being priced too high is quite a common one. Judging the correct value of a temporary resource is quite difficult and relies on understanding the market comprehensively. This level of knowledge is hard to gain and would be a problem in a Grid computing environment. The last problem is that of wealthy companies outbidding smaller companies. To prevent this, brokers can be allowed to participate only in a certain number of auctions over a set time frame.

B. Dutch Auction

In the Dutch auction, depicted in Fig. 7, the auctioneer starts the price of the item high and continually lowers the price until a buyer steps in and takes the item at that price.

The Dutch auction, though conceptually similar to the English auction, has several key differences. An important advantage is that market forces of supply-and-demand play a greater role in this auction. When a service is in demand, bidders will bid earlier and hence, the service fetches a higher price. When there are excess services available against the demand, people will not bid until the price drops until it is more reasonable. The suppliers also price their services accordingly.

Though there is less communication than the English auction model, it is still at a high level. Every time the item decreases in value, an announcement is broadcast to each bidder creating a lot of communication. Pricing difficulties are not as severe as in the English model as market forces will drive prices down or up. Overvalued resources therefore, do not pose a problem. However, there is still potential for undervaluing a resource. Like the English auction, buyers in the

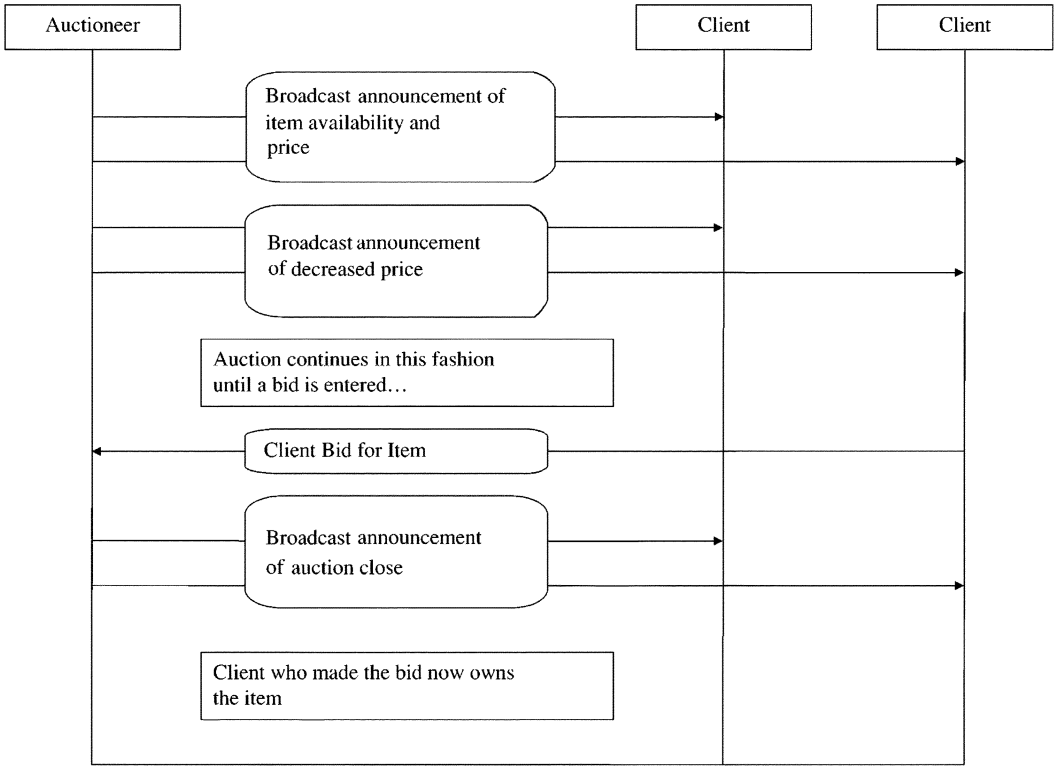


Fig. 7. The Dutch Auction Interaction Model.

Dutch auction have to judge the correct value of a service. This requires a knowledge of the market and current trends which is quite hard to come by.

C. Double Auction

The Double auction is very different from either of the previous two models. The Double auction works on a system of asks and bids. The sellers ask a price for their resources and the buyers make bids for the same. The auction therefore, is a continual process where the auctioneer matches up corresponding asks and bids to make a sale. The Double auction within a Grid context is shown in Fig. 8.

Within the Double auction, the presence of multiple simultaneous bidders for multiple sets of resources increases the throughput of the model. In a Grid computing environment, this allows one GMA to service all the GSPs and GRBs quickly and efficiently. The low levels of communication are also a positive feature. Both the seller and the bidder are only required to post the ask price and the bid price once and a message is sent back to them by the GMA announcing whether they have made a trade or not. This is much lower than in the previous models. Supply and demand are also factored into this model as trades can only be made with corresponding asks and bids. This means that what is perceived to be unreasonable will not sell.

Since this is a closed auction, it allows for private bidding but also keeps a lot of people in the dark. An unscrupulous GMA could exploit this and match similar bids and asks but keep the difference for himself. However, a system where the GMA matches bids and asks but does not carry out the transaction would be safe and this is likely to be the case in a Grid computing environment.

Table 9 Comparison Between Auction Models for Grid Environment

Auction Type	Social Welfare (Global Good)	Computational Efficieny	Communication Efficiency
English Auction	Resource is allocated to the GRB who values it the most.	Large amount of computational time spent in bidding process.	Very high levels of communication to allocate one resource. Low level of communication when demand is low.
Dutch Auction	Resource not necessarily given to the GRB who values it the most.	Less computational time required than for the English auction	Less communication per resource allocation than English auction. Higher level of communication per resource than English auction when demand is low.
Double Auction	GSPs do not have the option of taking the highest bidder during periods of high demand. Resource not necessarily given to the GRB who values it the most.	Smaller amount of computational time required per resource allocation than for the English or Dutch auction.	Smallest amount of communication per resource allocation out of the three auction models. No communication across the grid if no GRBs require resources.

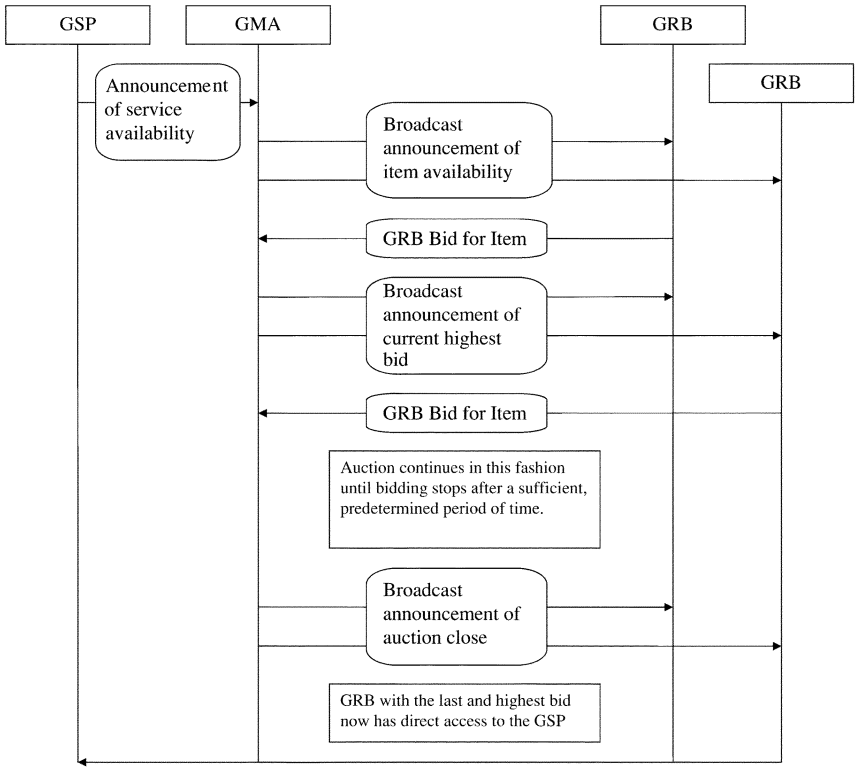


Fig. 8. The Double Auction Interaction Model.

D. Comparison and Comments

The three auction models are best compared for efficiency using some of the criteria stated in Section II. This comparison is summarized in Table 9. The Double auction uses an auction to imitate the traditional resource allocation methods. Instead of broadcasting to the whole grid when a resource becomes free for use, the GSPs only inform the GMA of the available resource. Similarly GRBs do not transmit their bids to the entire grid, just the GMA. The end result is that there is a reduction in the amount of network traffic per resource allocation and no network traffic at all when none of the GRBs require any resources.

Auctions provide a fair way of allocating resources in a grid environment despite the disadvantage of holding a resource while waiting for others. They are effective when dealing with a large number of participants and while dealing with objects whose value vary with perception.

VII. Summary and Conclusion

We have introduced computational economy as a model for tackling challenges of resource management within large-scale Grids and have discussed various approaches followed by representative works. We have proposed and discussed a reference system architecture driven by Grid economy. The use of computational economy within Nimrod-G and Gridbus brokers for compute and data intensive applications, respectively, has been presented. We have also formulated and evaluated scheduling in computational and data Grids environments. The results demonstrate effectiveness of commodity market-based resource allocation and also meet users' QoS requirements. In addition, auction models and their implications when applied to resource trading within Grid environments have been discussed.

We believe the support for economy-based resource management within Grid computing environments is essential for pushing Grids into mainstream computing. Therefore, we recommend that next-generation grids should consider economic incentive as one of the key design parameters. The importance of this has also been recognized by many national and international research agencies including the European Commission, which has identified Grid economy as one of the key thrust areas for research and development within their Next Generation Grids 2005–2010 program [19].

Acknowledgment

The authors would like to thank J. Giddy and H. Stockinger who contributed to some of the work described in this paper. R. Buyya would like to thank his Grid course students, T. Brophy and C. Talyor, who analyzed auction protocols and created the interaction diagrams based on material described in [53]. Some of the material included in this paper has been drawn from [50]–[56]. Nimrod-G has been the work of a number of people, including R. Sosic, J. Giddy, S. Garic, and C. Enticott. The Nimrod-G project has been supported by the Co-operative Research Center for Enterprise Distributed Systems (DSTC) since 1994. The contributors to the Gridbus Project include J. Yu, M. Placek, A. Barmouta, A. Sulistio, C. S. Yeo, R. Ranjan, D. C. Hong, H. Gibbins, and A. Luther. The Gridbus Project research and innovation sponsors include Australian Research Council, Storage Technology Corporation, Sun Microsystems, VPAC, IBM, and Singapore Computer Systems.

REFERENCES

[1] A. Barmouta and R. Buyya, "GridBank: A grid accounting services architecture (GASA) for distributed systems sharing and integration," presented at the Workshop Internet Computing and E-Commerce, Proc. 17th Annu. Int. Parallel and Distributed Processing Symposium (IPDPS 2003), Nice, France, Apr. 22–26, 2003.

[2] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke, "The data grid: Toward an architecture for the distributed management and analysis of large scientific datasets," *J. Netw. Comput. Applicat.*, vol. 23, no. 3, Jul. 2000.

[3] A. Lazar and N. Semret, "Auctions for Network Resource Sharing," Columbia Univ., TR 468-97-02, Feb. 1997.

[4] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal, *High Performance Computing: Paradigm and Infrastructure*, L. Yang and M. Guo, Eds, New York: Wiley. Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework, Fall 2004. (in print).

[5] B. Carpenter, IPv6 and the Future of the Internet, Jul., 23 2001. The Internet Society Member Briefing.

[6] B. Chun and D. Culler, "Market-Based Proportional Resource Sharing for Clusters," University of California, Berkeley, Missouri, USA, Technical Report CSD-1092, Jan. 2000.

[7] B. Chun, "Market-based cluster resource management," Ph.D. dissertation, University of California at Berkeley, Berkeley, CA, Oct. 2001.

[8] B. Cooper and H. Garcia-Molina, "Bidding for storage space in a peer-to-peer data preservation system," in *Proc. 22nd International Conference on Distributed Computing Systems (ICDSC 2002)*, Vienna, Austria, Jul. 2–5, 2002, http://www-db.stanford.edu/peers/.

[9] C. Kenyon and G. Cheliotis, "Architecture requirements for commercializing grid resources," presented at the 11th IEEE Int. Symp. High Performance and Distributed Computing (HPDC-11), Edinburgh, U.K., Jul. 2002.

[10] C. Waldspurger, T. Hogg, B. Huberman, J. Kephart, and W. Stornetta, "Spawn: A distributed computational economy," *IEEE Trans. Softw. Eng.*, vol. 18, no. 2, pp. 103–117, Feb. 1992.

[11] D. Abramson, J. Giddy, and L. Kotler, "High performance parametric modeling with nimrod/G: Killer application for the global grid?," presented at the Int. Parallel and Distributed Processing Symposium (IPDPS 2000), Cancun, Mexico, May 1–5, 2000.

[12] D. Abramson, P. Roe, L. Kotler, and D. Mather, "ActiveSheets: Super-computing with spreadsheets," presented at the High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conf., Seattle, WA, Apr. 2001.

[13] D. Abramson, R. Sosic, J. Giddy, and B. Hall, "Nimrod: A tool for performing parametised simulations using distributed workstations," presented at the 4th IEEE Int. Symp. High Performance Distributed Computing, Pentagon City, VA, Aug. 1995.

[14] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford, "Xenoservers: Accounted execution of untrusted code," presented at the 7th Workshop Hot Topics in Operating Systems (HotOS-VII), Rio Rico, AZ, Mar. 28–30, 1999.

[15] F. Berman and R. Wolski, "The AppLeS project: A status report," presented at the 8th NEC Research Symp., Berlin, Germany, May 1997.

[16] F. Berman, G. Fox, and T. Hey, *The Grid: Past, Present, Future, Grid Computing: Making the Global Infrastructure a Reality*. New York: Wiley, 2003.

[17] G. Fedak, C. Germain, V. Néri, and F. Cappello, "XtremWeb : A generic global computing system," presented at the 1st IEEE/ACM Int. Symp. Cluster Computing and the Grid (CCGrid 2001), Brisbane, Australia, May 15–18, 2001.

[18] G. Heiser, F. Lam, and S. Russell, "Resource management in the mungi single-address-space operating system," presented at the Australasian Computer Science Conf., Perth, Australia, Feb. 4–6, 1998.

[19] H. Bal *et al.*, "Next Generation Grid(s) European Grid Research 2005–2010," Information Society Technologies, European Commission, Expert Group Rep., Jun. 16, 2003.

[20] H. Casanova and J. Dongarra, "NetSolve: A network server for solving computational science problems," *Int. J. Supercomputing Applicat. High Performance Computing*, vol. 11, no. 3, pp. 212–223, 1997.

[21] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, "The AppLeS parameter sweep template: User-level middleware for the grid," presented at the IEEE Supercomputing Conf. (SC 2000), Dallas, TX, Nov. 2000.

[22] H. Stockinger, K. Stockinger, E. Schikuta, and I. Willers, "Toward a cost model for distributed and replicated data stores," presented at the 9th Euromicro Workshop on Parallel and Distributed Processing PDP 2001, Mantova, Italy, Feb. 2001.

[23] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a Future Computing Infrastructure*. San Mateo, CA: Morgan Kaufmann, 1999.

[24] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *Int. J. Supercomput. Applicat.*, vol. 11, no. 2, pp. 115–128, 1997.

[25] I. Foster, A. Roy, and V. Sander, "A quality of service architecture that combines resource reservation and application adaptation," presented at the IEEE/IFIP 8th Int. Workshop Quality of Service (IWQOS 2000), Pittsburgh, PA, Jun. 2000.

[26] Electricity Trading Over the Internet Begins in Six New England States (1999, May 13). [Online]. Available: http://industry.java.sun.com/javanews/stories/story2/0,1072,15093,00.html

[27] J. Almond and D. Snelling, "UNICORE: Uniform access to supercomputing as an element of electronic commerce," *Future Generation Comput. Syst.*, vol. 613, pp. 1–10, 1999.

[28] J. Bredin, D. Kotz, and D. Rus, "Utility Driven Mobile-Agent Scheduling," Dartmouth College, Hanover, NH, Tech. Rep. CS-TR98-331, Oct. 3, 1998.

[29] J. Brooke, M. Foster, S. Pickles, K. Taylor, and T. Hewitt, "Minigrids: Effective test-beds for grid application," presented at the 1st IEEE/ACM Int. Workshop Grid Computing (GRID 2000), Bangalore, India, Dec. 17, 2000.

[30] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and R. Buyya, "Libra: A computational economy based job scheduling system for clusters," in *Int. J. Software: Practice and Experience*, May 2004, vol. 34, pp. 573–590.

[31] J. Yu, S. Venugopal, and R. Buyya, "A Market-Oriented Grid Directory Service for Publication and Discovery of Grid Service Providers and their Services," Grid Computing and Distributed Systems (GRIDS) Laboratory, The University of Melbourne, Australia, Tech. Rep., GRIDS-TR-2003-0, Jan. 2003.

[32] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A resource management architecture for metacomputing systems," presented at the 4th Int. Workshop Job Scheduling Strategies for Parallel Processing, Orlando, FL, Mar. 30, 1998.

[33] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. (2004, Mar.) From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution. [Online]. Available: http://www.globus.org/wsrf/

[34] K. Ranganathan and I. Foster, "Decoupling computation and data scheduling in distributed data-intensive applications," presented at the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), Edinburgh, U.K., Jul. 2002.

[35] L. W. McKnight and J. Boroumand, "Pricing internet services: Approaches and challenges," *IEEE Comput.*, vol. 33, no. 2, pp. 128–129, Feb. 2000.

[36] Large Hadron Collider, CERN, Accessed (2004, Jan.). [Online]. Available: http://lhc-new-homepage.web.cern.ch/lhc-new-homepage/

[37] M. Baker and G. Fox, *High Performance Cluster Computing: Architectures and Systems*, R. Buyya, Ed. Upper Saddle River, NJ: Prentice-Hall PTR, 1999, vol. 1, Metacomputing: Harnessing informal supercomputers.

[38] M. Faerman, A. Su, R. Wolski, and F. Berman, "Adaptive performance prediction for distributed data-intensive applications," presented at the Supercomputing, Portland, OR, Nov. 1999.

[39] M. Frank. (2002, Apr.) *The OCEAN Project: The Open Computation Exchange & Auctioning Network* [Online]. Available: http://www.cise.ufl.edu/research/ocean/

[40] M. Huhns and L. Stephens, "Mutiagent systems and societies of agents," in *Multiagent Systems*, G. Weiss, Ed. Cambridge, MA: The MIT Press, 2000.

[41] M. Litzkow, M. Livny, and M. Mutka, "Condor—A hunter of idle workstations," presented at the 8th Int. Conf. Distributed Computing Systems (ICDCS 1988), San Jose, CA, Jan. 1988.

[42] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *J. Parallel Distributed Computing*, vol. 59, no. 2, pp. 107–131, Nov. 1999.

[43] M. Miller and K. Drexler, "Markets and computation: Agoric open systems," in *The Ecology of Computation*, B. Huberman, Ed. Amsterdam, The Netherlands: Elsevier, 1998.

[44] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin, "An economic paradigm for query processing and data migration in mariposa," in *3rd Int. Conf. Parallel and Distributed Information Systems*, Austin, TX, Sep., 28–30 1994.

[45] (2001, Jun.). Mojo Nation. [Online]. Available: http://www.mojonation.net/

[46] (2004, Jul.) MONARC Project, CalTech. [Online]. Available: http://monarc.cacr.caltech.edu/
[47] N. Kapadia and J. Fortes, "PUNCH: An architecture for web-enabled wide-area network-computing," *Cluster Computing: J. Netw., Softw. Tools Applicat.*, vol. 2, no. 2, pp. 153–164, Sep. 1999.
[48] N. Nisan, S. London, O. Regev, and N. Camiel, "Globally distributed computation over the internet: The POPCORN project," presented at the Int. Conf. Distributed Computing Systems (ICDCS'98), Amsterdam, The Netherlands, May 26–29, 1998.
[49] R. Buyya and M. Murshed, "GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing," in *J. Concurrency and Computation: Practic-Experience (CCPE)*, Nov.–Dec. 2002, vol. 14, pp. 1175–1220.
[50] R. Buyya, D. Abramson, and J. Giddy, "A case for economy grid architecture for service-oriented grid computing," presented at the Int, Parallel and Distributed Processing Symposium: 10th IEEE Int. Heterogeneous Computing Workshop (HCW 2001), San Francisco, CA, Apr. 23, 2001.
[51] ——, "An economy driven resource management architecture for global computational power grids," presented at the 2000 Int. Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), Las Vegas, NV, Jun. 26–29, 2000.
[52] ——, "Nimrod-G: An architecture for a resource management and scheduling system in a global computational grid," presented at the 4th Int. Conf. High Performance Computing in Asia-Pacific Region (HPC Asia 2000), Beijing, China, May 2000.
[53] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, "Economic models for resource management and scheduling in grid computing," *J. Concurrency Computation: Practice Experience (CCPE)*, vol. 14, no. 13–15, pp. 1507–1542, 2002.
[54] R. Buyya, H. Stockinger, J. Giddy, and D. Abramson, "Economic models for management of resources in peer-to-peer and grid computing," presented at the Int. Conf. Commercial Applications for High-Performance Computing, Denver, CO, Aug. 20–24, 2001.
[55] R. Buyya. (1999–2004) The World-Wide Grid (WWG). [Online]. Available: http://www.buyya.com/ecogrid/wwg/
[56] ——, "Economic-based distributed resource management and scheduling for Grid computing," Ph.D. thesis, Monash Univ., Melbourne, Victoria, Australia, Apr. 12, 2002.
[57] R. Buyya and S. Venugopal, "The gridbus toolkit for service oriented grid and utility computing: An overview and status report," presented at the 1st IEEE Int. Workshop Grid Economics and Business Models (GECON 2004), Seoul, Korea, Apr. 23, 2004.
[58] R. Cocchi, S. Shanker, D. Estrin, and L. Zhang, "Pricing in computer networks: Motivation, formulation, and example," *IEEE/ACM Trans. Networking*, vol. 1, no. 6, pp. 614–627, Dec. 1993.
[59] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling high bandwidth aggregates in the network," *Comput. Commun. Rev.*, vol. 32, no. 3, pp. 62–73, Jul. 2002.
[60] R. Moore, C. Baru, R. Marciano, A. Rajasekar, and M. Wan, "The grid: Blueprint for a new computing infrastructure," *Data Intensive Computing*, pp. 105–131, 1998.
[61] R. Smith and R. Davis, "The contract net protocol: High level communication and control in a distributed problem solver," *IEEE Trans. Comput.*, vol. C-29, no. 12, pp. 1104–1113, Dec. 1980.
[62] R. Wolski, J. Plank, J. Brevik, and T. Bryan, "Analyzing market-based resource allocation strategies for the computational grid," *Int. J. High-Performance Computing Applicat.*, vol. 15, no. 3, Fall 2001.
[63] R. Wolski, N. Spring, and J. Hayes, "The network weather service: A distributed resource performance forecasting service for metacomputing," *Future Generation Computing Syst.*, vol. 15, no. 5–6, pp. 757–768.
[64] S. Chapin, J. Karpovich, and A. Grimshaw, "The legion resource management system," presented at the 5th Workshop Job Scheduling Strategies for Parallel Processing, San Juan, Puerto Rico, Apr. 16, 1999.
[65] S. Jackson, *QBank: A Resource Management Package for Parallel Computers*. Richland, WA: Pacific Northwest National Laboratory, 2000.
[66] S. Lalis and A. Karipidis, "An open market-based framework for distributed computing over the internet," presented at the 1st IEEE/ACM Int. Workshop Grid Computing (GRID 2000), Bangalore, India, Dec. 17, 2000.
[67] S. Vazhkudai, S. Tuecke, and I. Foster, "Replica selection in the globus data grid," in *1st IEEE/ACM Int. Conf. Cluster Computing and the Grid (CCGRID 2001)*, Brisbane, Australia, May 15–18, 2001.
[68] S. Venugopal, R. Buyya, and L. Winton, "A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids," Grid Computing and Distributed Systems Lab., Univ. Melbourne, Melbourne, Australia, Tech. Rep., GRIDS-TR-2004-1, Feb. 2004.
[69] T. Sandholm, "Distributed rational decision making," in *Multi-Agent Systems: A Modern Introduction to Distributed Artificial Intelligence*, G. Weiss, Ed. Cambridge, MA: The MIT Press, 2000.
[70] W. H. Bell, D. G. Cameron, R. Carvajal-Schiaffino, A. P. Millar, K. Stockinger, and F. Zini, "Evaluation of an economy-based file replication strategy for a data grid," presented at the Int. Workshop Agent-Based Cluster and Grid Computing at CCGrid, Tokyo, Japan, May, 2003.
[71] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger, "Data management in an international data grid project," presented at the 1st Int. Workshop Grid Computing (Grid 2000), Bangalore, India, 2000.
[72] Y. Amir, B. Awerbuch, and R. S. Borgstrom, "A cost-benefit framework for online management of a metacomputing system," presented at the 1st Int. Conf. Information and Computational Economy, Charleston, SC, Oct. 25–28, 1998.
[73] Y. Amir, B. Awerbuch, A. Barak A., S. Borgstrom, and A. Keren, "An opportunity cost approach for job assignment in a scalable computing cluster," in *IEEE Trans. Parallel Distributed Syst.*, Jul. 2000, vol. 11, pp. 760–768.



Rajkumar Buyya is a Senior Lecturer and the Storage Tek Fellow for Grid Computing in the Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia. He is also serving as the Director of the Grid Computing and Distributed Systems Laboratory. He has authored/co-authored over 100 papers and technical documents that include three books—Microprocessor $\times 86$ Programming, Mastering C++, and Design of PARAS Microkernel.

Mr. Buyya was awarded the Dharma Ratnakara Memorial Trust Gold Medal for academic excellence by Mysore and Kuvempu Universities.



David Abramson is a Professor of Computer Science in the School of Computer Science and Software Engineering (CSSE) at Monash University, Victoria, Australia. He is a project leader in the Co-operative Research Centre for Distributed Systems Nimrod Project and also Chief Investigator on two ARC funded research projects. He has published over 130 papers and technical documents. He is a co-founder of Active Tools P/L with Dr. R. Sosic, a company which was established to commercialize the Nimrod project and Guardsoft, a company focused on commercializing the Guard project. The commercial version of Nimrod is now called EnFuzion and is marketed by Axceleon Inc.



Srikumar Venugopal is working towards the Ph.D. degree in the Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia.

He is a Research Assistant in the Grid Computing and Distributed Systems Laboratory, University of Melbourne. He is the lead Researcher and Developer of the Gridbus Grid Service Broker and has assisted in its usage in grid-enabling several scientific applications. His research interests include grid economy, data grids, scheduling, and mobile agents.

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; **35**:491–512
Published online 31 January 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.646



Scheduling parameter sweep applications on global Grids: a deadline and budget constrained cost–time optimization algorithm

Rajkumar Buyya[1,*,†], Manzur Murshed[2], David Abramson[3] and Srikumar Venugopal[1]

[1]*Grid Computing and Distributed Systems Laboratory, Department of Computer Science and Software Engineering, The University of Melbourne, Parkville, Melbourne, VIC 3010, Australia*
[2]*Gippsland School of Computing and Information Technology, Monash University, Gippsland Campus, Churchill, VIC 3842, Australia*
[3]*School of Computer Science and Software Engineering, Monash University, Caulfield Campus, Melbourne, VIC 3145, Australia*

SUMMARY

Computational Grids and peer-to-peer (P2P) networks enable the sharing, selection, and aggregation of geographically distributed resources for solving large-scale problems in science, engineering, and commerce. The management and composition of resources and services for scheduling applications, however, becomes a complex undertaking. We have proposed a computational economy framework for regulating the supply of and demand for resources and allocating them for applications based on the users' quality-of-service requirements. The framework requires economy-driven deadline- and budget-constrained (DBC) scheduling algorithms for allocating resources to application jobs in such a way that the users' requirements are met. In this paper, we propose a new scheduling algorithm, called the DBC cost–time optimization scheduling algorithm, that aims not only to optimize cost, but also time when possible. The performance of the cost–time optimization scheduling algorithm has been evaluated through extensive simulation and empirical studies for deploying parameter sweep applications on global Grids. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: Grid computing; scheduling; computational economy; Nimrod-G; Gridbus broker

*Correspondence to: Rajkumar Buyya, Grid Computing and Distributed Systems Laboratory, Department of Computer Science and Software Engineering, The University of Melbourne, Parkville, Melbourne, VIC 3010, Australia.
†E-mail: raj@cs.mu.oz.au

Copyright © 2005 John Wiley & Sons, Ltd.

Received 19 December 2003
Revised 3 June 2004
Accepted 23 August 2004

1. INTRODUCTION

Computational Grids [1] and *peer-to-peer* (P2P) computing [2] networks are emerging as next-generation parallel and distributed computing platforms for solving large-scale computational and data-intensive problems in science, engineering, and commerce. They enable the *sharing*, *selection*, and *aggregation* of a wide variety of geographically distributed resources including supercomputers, storage systems, databases, data sources, and specialized devices owned by different organizations. However, resource management and application scheduling is a complex undertaking due to large-scale heterogeneity in resources, management policies, users, and application requirements in these environments [3].

A typical world-wide Grid computing environment is shown in Figure 1. In such a Grid marketplace and economy, the two key players are: *resource owners* (Grid service providers) and *end users* (Grid service consumers). The *resource owners* and *consumers/end-users* have different goals, objectives, strategies, and demand patterns. The resources are heterogeneous in terms of their architecture, power, configuration, and availability. They are owned and managed by different organizations with different access policies and cost models that vary with time, users, and priorities. Different applications have different computational models that vary with the nature of the problem.

In our earlier work [4–7], we investigated the use of economics as a metaphor for management of resources and scheduling applications in Grid computing environments. The computational economy framework provides a mechanism for regulating the supply of and demand for resources and allocating them to applications based on the users' quality-of-services (QoS) requirements [3]. It also offers an incentive to resource owners for sharing resources on the Grid, and offers end users a trade-off between the timeframe for result delivery and computational expenses.

A Grid scheduler, often called *resource broker*, acts as an interface between the user and distributed resources and hides the complexities of Grid computing [4,5]. It performs resource discovery, negotiates for access costs using trading services, maps jobs to resources (*scheduling*), stages the application and data for processing (*deployment*), starts job execution, and finally gathers the results. It is also responsible for monitoring and tracking the progress of application execution along with adapting to the changes in the runtime environment of the Grid, variation in resource share availability, and failures. Essentially, the Grid broker does application scheduling on distributed Grid resources on which it does not have full control—the local scheduler has its own policies and performs actual allocation of resource(s) to the user job(s).

The previous work in scheduling on distributed systems such as clusters and supercomputers has focused on extracting the maximum throughput from the entire system [8,9]. Grid scheduling, as shown by some of the related works in Section 2, concentrates on improving response times in an environment containing *autonomous* resources whose availability dynamically varies with time. The Grid scheduler has to interact with the local schedulers managing computational resources and adapt its behavior to changing resource loads. Thus the scheduling is conducted from the perspective of the application or the user rather than that of the system.

In our Grid economy framework, the resource brokers use economy-driven *deadline and budget-constrained* (DBC) scheduling algorithms for allocating resources to application jobs in such a way that the users' requirements are met. In our early work [7], we developed three scheduling algorithms for cost, time, and time-variant optimization strategies that support deadline and budget constraints.

Copyright © 2005 John Wiley & Sons, Ltd.

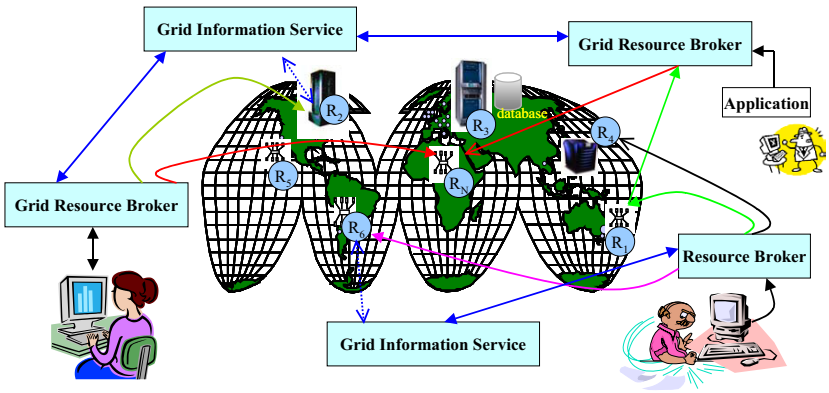


Figure 1. A generic view of the World-Wide Grid computing environment.

We implemented them within the Nimrod-G broker and explored their capability for scheduling task-farming or parameter-sweep applications such as drug design [10] on the World-Wide Grid (WWG) testbed resources [8]. To meet users' QoS requirements, the broker leases Grid resources and services dynamically at runtime depending on their capability, cost, and availability.

In this work, we propose a new scheduling algorithm, called *DBC cost–time optimization*, which extends the DBC cost-optimization algorithm to optimize for time, keeping the cost of computation at the minimum. Resources with the same cost are grouped together and a time-optimization scheduling strategy is applied while allocating jobs to a group. We demonstrate the ability of this new scheduling algorithm by implementing it within the economic Grid resource broker simulator built using the GridSim toolkit [11]. The performance of this new algorithm is evaluated by scheduling a synthetic task-farming application on simulated WWG testbed resources for different deadline and budget scenarios. We then compare and contrast the results of scheduling with the cost-optimization algorithm.

The rest of this paper is organized as follows. The related works in Grid resource management and scheduling are discussed in Section 2. Issues in scheduling applications on Grids along with a new DBC cost–time optimization scheduling algorithm proposed in this paper are discussed in Section 3. An economy-driven grid resource broker simulated using the GridSim toolkit and its internal components that simulate and manage the execution of task-farming applications are presented in Section 4. The simulation of heterogeneous resources with different capabilities and access costs, the creation of a synthetic application, and the evaluation, by simulation, of the proposed cost–time optimization scheduling algorithm versus the cost optimization algorithm is discussed in Section 5. The proposed algorithm is evaluated by empirical studies in Section 6. The final section presents the summary and conclusion.

2. RELATED WORK

A number of projects are investigating scheduling on distributed systems [3]. They include Grid resource management and scheduling systems such as Condor [12,13], Globus [14], Legion [15],

Copyright © 2005 John Wiley & Sons, Ltd.

AppLeS [16], NetSolve [17], and DISCWorld [18], which use system-centric scheduling strategies, and REXEC [19] and Spawn [20], which support computational economy-based resource management within cluster computing environments. The interest in computational economy is rapidly increasing within the Grid community [21] and projects such as G-Commerce [22] are exploring competitive resource pricing and allocation issues within the Grid environments.

The Application-Level Scheduling (AppLeS) project builds agents for each application responsible for offering a scheduling mechanism [16]. It uses a system-centric scheduling policy, which is targeted at minimizing the completion time—it does not take account of the economic cost of jobs processing while selecting resources. A recently developed AppLeS parameter-sweep template (APST) also uses system-centric scheduling strategies [23].

NetSolve is a client–agent–server system, which enables the user to solve complex scientific problems remotely [17]. The NetSolve agent does the scheduling by searching for those resources that offer the best performance in a network. The applications need to be built using one of the application programming interfaces (APIs) provided by NetSolve to perform remote procedure call (RPC)-like computations. NetSolve also provides an API for creating *task-farming* applications. The scheduling system maps jobs on resources that have appropriate libraries without taking the cost/price for processing jobs on them into consideration. Ninf [24] is an RPC implementation for grid computing similar to NetSolve.

Distributed Information Systems Control World (DISCWorld) is a service-oriented metacomputing environment, based on the client–server–server model [18]. Remote users can log in to this environment over the Internet and request access to data, and also invoke services or operations on the available data. DISCWorld aims for remote information access. The scheduling strategies used in the DISCWorld system are also system-centric in nature.

Another related tool that uses the concept of computational economy is REXEC, a remote execution environment [19] for co-operative distributed systems such as clusters with a centralized scheduling manager. At the command line, the maximum rate (credits per minute) that the user is willing to pay for CPU time can be specified. The REXEC client selects a node that fits the user's requirements and executes the application on it. The REXEC system provides an extended shell for remote execution of applications on clusters. Its scheduling strategies are targeted for centralized systems, and the allocation of resource share is proportional to the users' valuation of their jobs. In our Grid resource broker, scheduling strategies are targeted for geographically distributed systems—each resource has its own scheduler that performs actual allocation of resources to user jobs. That means cluster schedulers use cooperative computing economy since they aim for global optimization, whereas Grid schedulers use competitive computing economy since every entity aims to optimize its own objectives.

Spawn [20] is one of the earliest computational-economies-based resource-allocation systems. It uses a second-price auction model where tasks have to bid for computational time. Allocation of funding for tasks has to be performed by the application-level manager, which needs to be implemented by a programmer along with the necessary allocation strategies and priorities. That is, Spawn has mostly focused on developing an auction-based infrastructure for the computing marketplace and has left it to the programmers (application developers) to take care of allocation policies and algorithms. In fact, this complements the work described in this paper, as our focus is on developing a computational-economy-based application-scheduling system in addition to providing a parameter-sweep application-creation framework.

Copyright © 2005 John Wiley & Sons, Ltd.

A number of works have explored the problem of deadline scheduling in real-time systems [25–27]. However, they mostly consider deadline parameters for individual jobs rather than groups. In addition, they have not addressed the concept of leasing third party services for meeting the deadline as they are restricted to centralized resource management schemes and single administrative domain resources.

3. DBC COST–TIME OPTIMIZATION SCHEDULING ALGORITHM

3.1. Application model and scheduling

The parameter-sweep application model has emerged as a 'killer application model' [4] for composing high-throughput computing (HTC) applications for processing on global Grids. This model is a combination of task and data parallel models and applications formulated using this model contain large number of independent jobs operating on different data sets. A range of scenarios and parameters to be explored are applied to the program input values to generate different data sets. The programming and execution model of such applications resembles the Single Program Multiple Data (SPMD) model. The execution model essentially involves processing N independent jobs (each with the same task specification, but a different dataset) on M distributed computers where N is, typically, much larger than M. Fortunately, this high-throughput parametric computing model is simple, yet powerful enough to formulate distributed execution of many application areas such as: radiation equipment calibration analysis [4], searching for extra-terrestrial intelligence [28], protein folding [29], molecular modeling for drug design [10], human-genome sequence analysis [30], brain activity analysis, high-energy physics events analysis [31], *ad hoc* network simulation [32], crash simulation, tomography [33], financial modeling, and Mcell simulations [34]. Therefore, high-throughput parametric computing is considered as the killer application for the Grid.

Scheduling and orchestrating the execution of parameter-sweep applications on world-wide distributed computers appears simple, but complexity arises when users place QoS constraints such as execution completion time (deadline) and computation cost (budget) limitations along with optimization parameter preference. Such a guarantee of service is hard to provide in a Grid environment since its resources are shared, heterogeneous, distributed in nature, and owned by different organizations which have their own policies and charging mechanisms. In addition, scheduling algorithms need to adapt to the changing load and resource availability conditions in the Grid in order to achieve performance and at the same time meet the deadline and budget constraints.

The integration of computational economy as part of a scheduling system greatly influences the way computational resources are selected to meet the users' requirements. A user should be able to submit their application along with their requirements to a scheduling system such as Nimrod-G, which can process the application on the Grid on the user's behalf and try to complete the assigned work within a given deadline and cost. The deadline represents a time by which the user requires the result, and is often imposed by external factors like production schedules or research deadlines.

3.2. A new scheduling algorithm

We have developed a number of algorithms for DBC scheduling of task-farming parameter-sweep applications on the Grid. They are: cost-optimization, time-optimization, and conservative time-optimization scheduling algorithms. The performance of these algorithms has been evaluated by

Copyright © 2005 John Wiley & Sons, Ltd.

implementing them within the Nimrod-G resource broker [11] for scheduling real-world applications and economic-broker simulators [7] through synthetic workloads.

In this paper, we propose a new DBC Grid scheduling algorithm, called the cost–time optimization scheduling algorithm, which builds on the cost-optimization and time-optimization scheduling algorithms. This is accomplished by applying the time-optimization algorithm to schedule task-farming application jobs on distributed resources having the same processing cost. A detailed algorithm for mapping jobs to resources based on this new strategy is listed in Figure 2. An application containing N jobs along with the deadline and budget constraints are passed as input parameters to the algorithm for processing on M distributed resources/machines. Essentially, the user passes these details to the Grid broker that leases resources dynamically at runtime depending on their availability, capability, cost, and users' QoS requirements. In the next subsection, we discuss methods for evaluation of this new algorithm, and later sections present detailed evaluation.

3.3. Evaluation methods and complexity analysis

A variety of techniques and technologies exist for carrying out performance evaluation of resource-management and scheduling algorithms. Some evaluation techniques include: analytical, simulation, emulation and empirical. Some of the notable Grid tools are SimGrid [35] and GridSim [11] for *simulation*; MicroGrid [36] for *emulation*; and Nimrod-G [5] that supports creation of pluggable schedulers for *empirical* evaluation. In addition, empirical evaluation needs system-level Grid middleware such as Globus for deploying jobs securely on Grid testbed resources.

Through analysis of the algorithm listed in Figure 2, it can be seen that the loop in step 4(e) determines the time complexity. This loop costs 1 when there is 1 resource group (i.e. when all M machines are of the same cost) and M when there are M resource groups (i.e. all machines have different costs). Within this loop, the loop in step 4(i) costs N. Within the second loop, the step leading to job matching costs is M for 1 resource group and 1 for M resource groups.

Bringing it all together, the overall complexity can be calculated as:

- $\mathrm{O}(1 * (N * (M))) = \mathrm{O}(NM)$ when all resources have the same cost; and
- $\mathrm{O}(M * (N * (1))) = O(NM)$ when all resources have different costs.

The cost-optimization algorithm also has a time complexity of $\mathrm{O}(NM)$. However, as we will show in later sections, the cost–time-optimization algorithm processes the jobs in less time than the cost-optimization algorithm while using the same amount of budget.

Further on in this paper, we use simulation and empirical methods to evaluate the performance of an economic-based resource allocation strategy. Simulation allows the creation of large-scale virtual Grid environments, and usage and availability scenarios that can be repeated and controlled. We have selected GridSim for simulation as it supports modeling and simulation of a variety of resources—shared and distributed memory machines, managed as time- or space-shared resources. To evaluate the feasibility of the proposed scheduling strategy, we have implemented a plug-in scheduler for the Nimrod-G Grid resource broker. However, it should be noted that in a real Grid testbed, it is impossible to conduct *repeatable* and *comparable* evaluations as the availability of resources varies with time and there is no centralized control to create a stable environment. Sections 5 and 6 present the results of evaluation through simulation and empirical methods respectively.

Copyright © 2005 John Wiley & Sons, Ltd.

Algorithm: *DSC_Scheduling_with_Cost_Time_Optimization* (*Application*: *N jobs*, *Resources*: *M*, *Deadline*: *D*, *Budget*: *B*)

(1) RESOURCE DISCOVERY: Identify characteristics, configuration, capability, and suitability of resources using the Grid information services (GIS).
(2) RESOURCE TRADING: Identify *the cost* of all resources and *the capability* to be delivered per cost-unit. The *resource cost* can be expressed in units such as processing cost per Million Instructions (MI) cost per job, CPU cost per time unit etc., and the scheduler needs to choose suitable units for comparison.
(3) If the user supplies *d* and *b* factors, then determine the absolute deadline and budget based on the capability of resources and their cost, and the application processing requirement (e.g. total MI required).
(4) SCHEDULING: Repeat while there exist *unprocessed jobs* and the current time and processing expenses are within the deadline and budget limits. [This step is triggered for each scheduling event or whenever a job completes. The event period is a function of deadline, job processing time, rescheduling overhead, resource share variation, etc.]
[SCHEDULE ADVISOR with Policy]

   (a) For each resource, predict and establish the *job consumption rate* or *the available resource share* through the measure and extrapolation strategy taking into account the time taken to process previous jobs.
   (b) SORT the resources by increasing order of *cost*. If two or more resources have the *same cost*, order them such that powerful ones (e.g. higher job consumption rate or resource share availability, but, the first time, based on the total theoretical capability, say the total Million Instructions per second (MIPS) are preferred first.
   (c) *Create resource groups* containing resources with the same cost.
   (d) If any of the resource has jobs assigned to it in the previous scheduling event, but not dispatched to the resource for execution and there is variation in resource availability, then move appropriate number of jobs to the Unassigned-Jobs-List. This helps in updating the whole schedule based on the latest resource availability information.
   (e) Repeat the following steps for each resource group as long as there exist unassigned jobs:
      (i) *Repeat the following steps for each job in the Unassigned-Jobs-List depending on the processing cost and the budget availability*: [It uses the *time optimization strategy*.]
         - Select a job from the Unassigned-Jobs-List.
         - For each resource, calculate/predict the job completion time taking into account previously assigned jobs and the job completion rate and resource share availability.
         - Identify the resource with the least completion time and assign the job to it provided it is able to complete the job within the deadline. Remove the assigned job from the Unassigned-Jobs-List.

(5) [DISPATCHER with Policy]
Repeat the following steps for each resource if it has jobs to be dispatched:

- Identify the number of jobs that can be submitted without overloading the resource. Our default policy is to dispatch jobs as long as the number of user jobs deployed (active or in queue) is less then the number of PEs (Processing Elements, i.e. CPUs) in the resource.

Figure 2. DBC scheduling with cost–time optimization.

Copyright © 2005 John Wiley & Sons, Ltd.

4. IMPLEMENTATION OF ECONOMIC GRID BROKER SIMULATOR

The GridSim toolkit is used to simulate a Grid environment and a Nimrod-G-like deadline and budget constrained scheduling system called economic Grid resource broker. The simulated Grid environment contains multiple resources and user entities with different requirements. The user and broker entities extend the GridSim class. All the users create experiments containing application specification requirements (a set of Gridlets that represent application jobs) and QoS requirements (deadline and budget constraints with optimization strategy). When the simulation starts, the user entity creates an instance of its own broker entity and passes a request for processing application jobs. We briefly discuss features of the GridSim toolkit and its usage in the implementation of the economic broker simulator that supports performance evaluation of scheduling algorithms.

4.1. GridSim: a Grid modeling and simulation toolkit

The GridSim toolkit provides a comprehensive facility for simulation of different classes of heterogeneous resources, users, applications, resource brokers, and schedulers [11]. It has facilities for the modeling and simulation of resources and network connectivity with different capabilities, configurations, and domains. It supports primitives for application composition, information services for resource discovery, and interfaces for assigning application tasks to resources and managing their execution. These features can be used to simulate resource brokers or Grid schedulers for evaluating performance of scheduling algorithms or heuristics. We have used the GridSim toolkit to create a resource broker that simulates Nimrod-G for design and evaluation of deadline and budget constrained scheduling algorithms with cost and time optimizations.

The GridSim toolkit resource modeling facilities are used to simulate the World-Wide Grid resources managed as time- or space-shared scheduling policies. The broker and user entities extend the GridSim class to inherit ability for communication with other entities. In GridSim, application tasks/jobs are modeled as *Gridlet* objects that contain all the information related to the job and its execution management details such as job length in MI, disk input/output operations, input and output file sizes, and the job originator. The broker uses GridSim's job management protocols and services to map a Gridlet to a resource and manage it throughout its lifecycle. The broker also maintains full details of application scheduling trace data both at coarse and fine levels, which can be used in performance analysis.

4.2. Economic Grid broker simulator architecture

The broker entity architecture and its interaction with other entities is shown in Figure 3. The key components of the broker are: experiment interface, resource discovery and trading, scheduling flow manager backed with scheduling heuristics and algorithms, Gridlets dispatcher, and Gridlets receptor. A detailed discussion of the broker implementation using the GridSim toolkit can be found in Buyya and Murshed [11]. However, to enable the understanding of the broker framework in which the new scheduling algorithm is implemented, we briefly present its operational model.

(1) The user entity creates an experiment that contains the application description (a list of Gridlets to be processed) and sends user requirements to the broker via the experiment interface.

Copyright © 2005 John Wiley & Sons, Ltd.

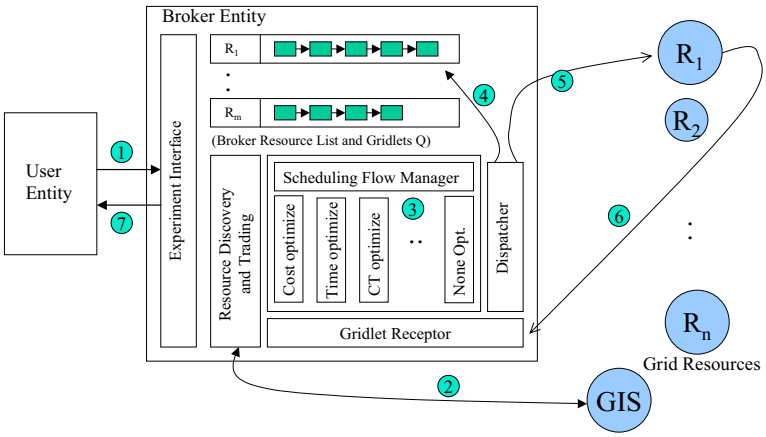


Figure 3. Economic Grid resource broker architecture and its interaction with other entities.

(2) The broker resource discovery and trading module interacts with the GridSim GIS entity to identify the contact information of the resources, and then interacts with resources to establish their configuration and access cost. It creates a broker resource list that acts as a placeholder for maintaining resource properties, a list of Gridlets committed for execution on the resource, and the resource performance data as predicted through the measurement and extrapolation methodology.
(3) The scheduling flow manager selects an appropriate scheduling algorithm for mapping Gridlets to resources depending on the user's requirements (deadline and budget limits, and optimization strategy—cost, cost–time, time, and conservative-time). Gridlets that are mapped to a specific resource are added to the Gridlets list in the broker resource. In this case, the broker selects the algorithm for DBC cost–time-optimization scheduling.
(4) For each of the resources, the dispatcher selects the number of Gridlets that can be staged for execution according to the usage policy to avoid overloading resources with single user jobs.
(5) The dispatcher then submits Gridlets to resources using the GridSim's asynchronous service.
(6) When the Gridlet processing completes, the resource returns it to the broker's Gridlet receptor module, which then measures and updates the runtime parameter, *resource or MI share available to the user*. It aids in predicting the job consumption rate for making scheduling decisions.
(7) Steps 3–6 continue until all the Gridlets are processed or the broker exceeds the deadline or budget limits. At the end, the broker returns updated experiment data along with processed Gridlets back to the user entity.

5. SIMULATION-BASED PERFORMANCE EVALUATION

To simulate and evaluate application scheduling in a GridSim environment using the economic Grid broker requires the modeling and creation of GridSim resources and applications that model jobs

Copyright © 2005 John Wiley & Sons, Ltd.

as Gridlets. In this section, we present resource and application modeling along with the results of scheduling experiments with QoS-driven application processing.

5.1. Resource modeling

We modeled and simulated a number of time- and space-shared resources with different characteristics, configurations, and capability as those in the World-Wide Grid testbed. We have selected the latest CPU models AlphaServer ES40, Sun Netra 20, Intel VC820 (800EB MHz, Pentium III), and SGI Origin 3200 1X 500 MHz R14k, released by their manufacturers Compaq, Sun, Intel, and SGI respectively. The processing capability of these PEs in simulation time-units is modeled after the base value of the SPEC CPU (INT) 2000 benchmark ratings [37]. To enable the users to model and express their application processing requirements in terms of MI or MIPS on the standard machine, we *assume the MIPS rating of PEs is same as the SPEC rating*.

Table I shows the characteristics of the resources simulated and their PE access cost per time unit in Grid dollars (G$). The PE capability of resources is derived from their actual SPEC rating and access cost in G$ is artificially assigned. The simulated resources resemble the World-Wide Grid testbed resources used in the Nimrod-G scheduling experiments [38]. The access cost of a PE in G$/time unit does not necessarily reflect the cost of processing when PEs have different capabilities. The brokers need to translate the access cost into the G$/MI for each resource. Such translation helps in identifying the relative cost of resources for processing Gridlets on them. It can be noted some of the resources in Table I have the same MIPS/G$, for example, R4 and R8.

5.2. Application modeling

We have modeled a task-farming application that consists of 200 jobs. In GridSim, these jobs are packaged as Gridlets whose contents include the job length in MI, the size of job input, and output data in bytes, along with various other execution-related parameters when they move between the broker and resources. The job length is expressed in terms of the time it takes to run on a standard resource PE with SPEC/MIPS rating of 100. Gridlets' processing time is expressed in such a way that they are expected to take at least 100 time units with a random variation of 0–10 per cent on the positive side of the standard resource. That means Gridlets' job length (processing requirements) can be at least 10 000 MI with a random variation of 0–10 per cent on the positive side. This 0–10 per cent random variation in Gridlets' job length is introduced to model heterogeneous tasks similar to those present in the real-world parameter-sweep applications.

5.3. Scheduling experiments with cost- and cost–time-optimization strategies

We performed both cost- and cost–time optimization scheduling experiments with different DBC values for a single user. The deadline is varied in simulation time from 100 to 3600 in steps of 500. The budget is varied from G$ 5000 to 22 000 in steps of 1000. The number of Gridlets processed, deadline utilized, and budget spent for the DBC cost-optimization scheduling strategy is shown in Figures 4(a), 4(c), and 4(e), and for the cost–time-optimization scheduling strategy in Figures 4(b), 4(d), and 4(f). In both cases, when the deadline is low (e.g. 100 time units), the number of Gridlets processed increases as the budget value increases. When a higher budget is available, the broker leases

Copyright © 2005 John Wiley & Sons, Ltd.

Table I. World-Wide Grid testbed resources simulated using GridSim.

Resource name in simulation	Reference resource characteristics: vendor, type, node OS, no. of PEs	Equivalent resource in World-Wide Grid (hostname, location)	A PE SPEC/MIPS rating	Resource manager type	Price (G$/PE time unit)	SPEC/MIPS per G$
R0	Compaq, AlphaServer, CPU, OSF1, 4	grendel.vpac.org, VPAC, Melb, Australia	515	Time-shared	8	64.37
R1	Sun, Ultra, Solaris, 4	hpc420.hpcc.jp, AIST, Tokyo, Japan	377	Time-shared	4	94.25
R2	Sun, Ultra, Solaris, 4	hpc420-1.hpcc.jp, AIST, Tokyo, Japan	377	Time-shared	3	125.66
R3	Sun, Ultra, Solaris, 2	hpc420-2.hpcc.jp, AIST, Tokyo, Japan	377	Time-shared	3	125.66
R4	Intel, Pentium/VC820, Linux, 2	barbera.cnuce.cnr.it, CNR, Pisa, Italy	380	Time-shared	1	380.0
R5	SGI, Origin 3200, IRIX, 6	onyx1.zib.de, ZIB, Berlin, Germany	410	Time-shared	5	82.0
R6	SGI, Origin 3200, IRIX, 16	Onyx3.zib.de, ZIB, Berlin, Germany	410	Time-shared	5	82.0
R7	SGI, Origin 3200, IRIX, 16	mat.ruk.cuni.cz, Charles U., Prague, Czech Republic	410	Space-shared	4	102.5
R8	Intel, Pentium/VC820, Linux, 2	marge.csm.port.ac.uk, Portsmouth, U.K.	380	Time-shared	1	380.0
R9	SGI, Origin 3200, IRIX, 4 (accessible)	green.cfs.ac.uk, Manchester, U.K.	410	Time-shared	6	68.33
R10	Sun, Ultra, Solaris, 8,	pitcairn.mcs.anl.gov, ANL, Chicago, U.S.A.	377	Time-shared	3	125.66

Copyright © 2005 John Wiley & Sons, Ltd.

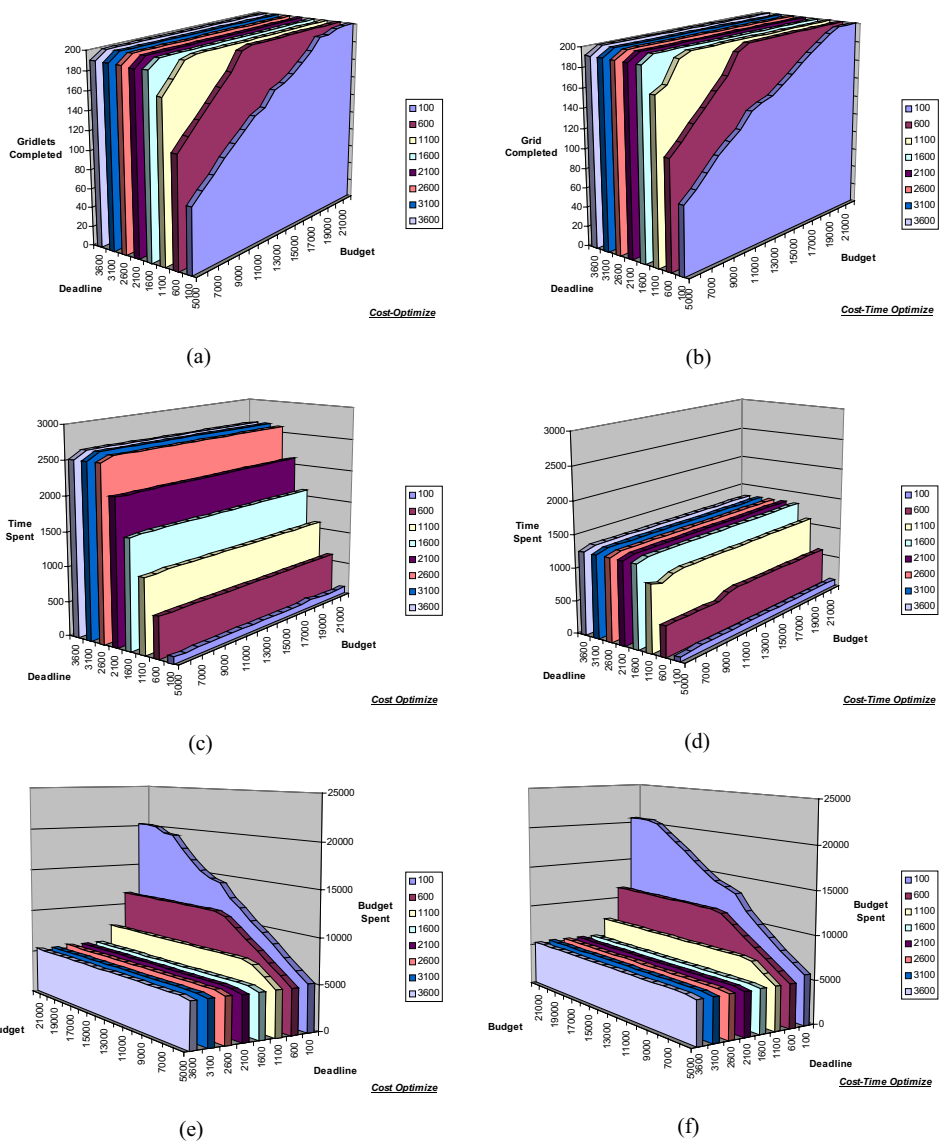


Figure 4. The number of Gridlets processed, time, and budget spent for different deadline and time limits when scheduled using the cost- and cost–time-optimization algorithms: (a) no. of Gridlets processed; (b) no. of Gridlets processed; (c) time spent for processing Gridlets; (d) time spent for processing Gridlets; (e) budget spent for processing Gridlets; (f) budget spent for processing Gridlets.

Copyright © 2005 John Wiley & Sons, Ltd.

expensive resources to process more jobs within the deadline. Alternatively, when scheduling with a low budget, the number of Gridlets processed increases as the deadline is relaxed.

The impact of budget for different values of deadline is shown in Figures 4(e) and 4(f) for cost and cost–time strategies. For a larger deadline value (see the time utilization for a deadline of 3600), the increase in budget value does not have much impact on resource selection. When the deadline is too tight (e.g. 100), it is likely that the complete budget is spent processing Gridlets within the deadline.

It can be observed that the number of Gridlets processed and the budget-spending pattern is similar for both scheduling strategies. However, the time spent for the completion of all the jobs is significantly different (see Figures 4(c) and 4(d)), as the deadline becomes relaxed. For deadline values from 100 to 1100, the completion time for both cases is similar, but as the deadline increases (e.g. from 1600 to 3600), the experiment completion time for the cost–time-scheduling optimization strategy is much less than that for the cost-optimization scheduling strategy. This is because when there are many resources with the same MIPS/G$, the cost–time-optimization scheduling strategy allocates jobs to them using the time-optimization strategy for the entire deadline duration since there is no need to spent extra budget for doing so. This does not happen in case of the cost-optimization strategy—it allocates as many jobs as the first cheapest resource can complete by the deadline, and then allocates the remaining jobs to the next cheapest resources.

A trace of resource selection and allocation using cost- and cost–time-optimization scheduling strategies, shown in Figure 5, indicates their impact on the application processing completion time. When the deadline is tight (e.g. 100), there is high demand for all the resources in a short time; the impact of cost- and cost–time-scheduling strategies on the completion time is similar as all the resources are used up so long as budget is available to process all jobs within the deadline (see Figures 5(a) and 5(b)). However, when the deadline is relaxed (e.g. 3100), it is likely that all jobs can be completed using the first few cheapest resources. In this experiment there were resources with the same cost and capability (e.g. R4 and R8). The cost-optimization strategy selected resource R4 to process all the jobs (see Figure 5(c)), whereas the cost–time-optimization strategy selected both R4 and R8 (see Figure 5(d)) since both resources cost the same price, and completed the experiment earlier than the cost-optimization scheduling (see Figures 4(c) and 4(d)). This situation can be observed clearly in scheduling experiments with a large budget for different deadline values (see Figure 6). Note that the left-most solid curve marked ‘All’ in the Resources axis in Figure 6 represents the aggregation of all resources.

As the deadline increases, the cost-optimization algorithm predominantly schedules jobs on the resource R4 (see Figure 6(a)), whereas the cost–time-optimization algorithm scheduled jobs on resources R4 and R8 (see Figure 6(b)), the first two cheapest resources with the same cost. Therefore, the application scheduling using the cost–time-optimization algorithm is able to finish earlier compared to the one scheduled using the cost-optimization algorithm (see Figure 7), and both strategies spend the same amount of budget processing their jobs (see Figure 8). The completion time for the cost-optimization scheduling continued to increase with increase of the deadline as the broker allocated more jobs to resource R4 and less to resource R8. However, the completion time for deadline values 3100 and 3600 was the same as 2600 since the broker allocated jobs to only resource R4. This was not the case with the cost–time-optimization scheduling since jobs were allocated proportionally to both resources R4 and R8 and thus minimizing the completion time without spending any extra budget.

A microscopic evaluation of mapping of jobs to different resources under the *single user* and the *multiple competing users* scenarios for the entire execution period can be found in [3].

Copyright © 2005 John Wiley & Sons, Ltd.

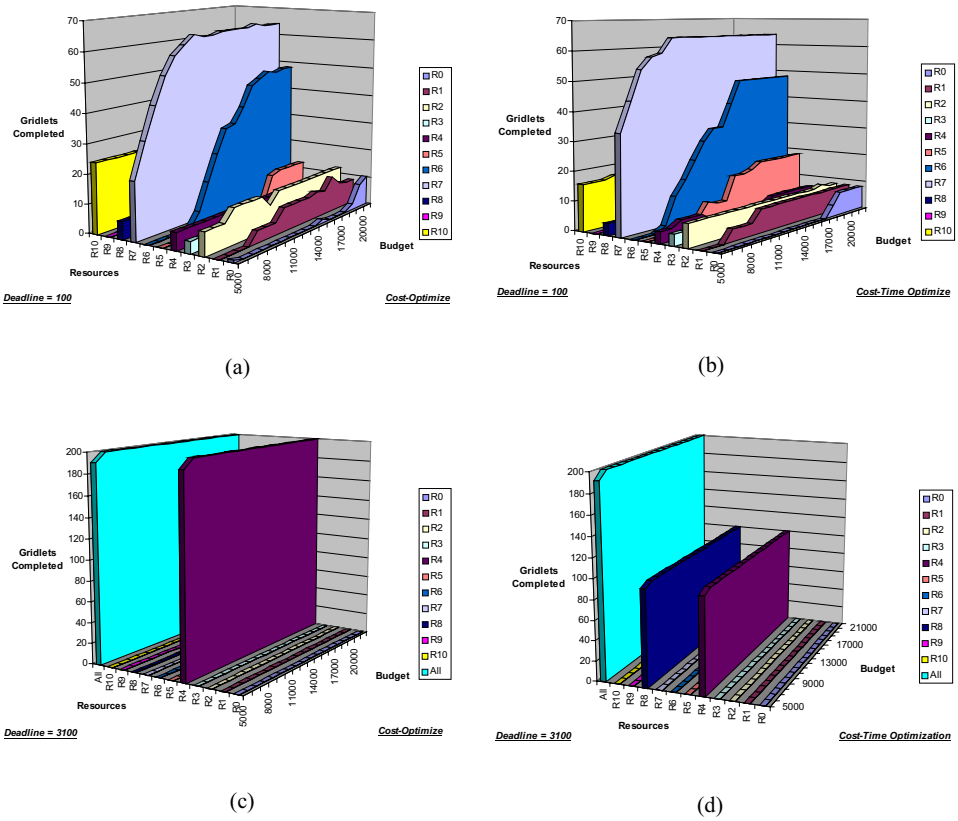


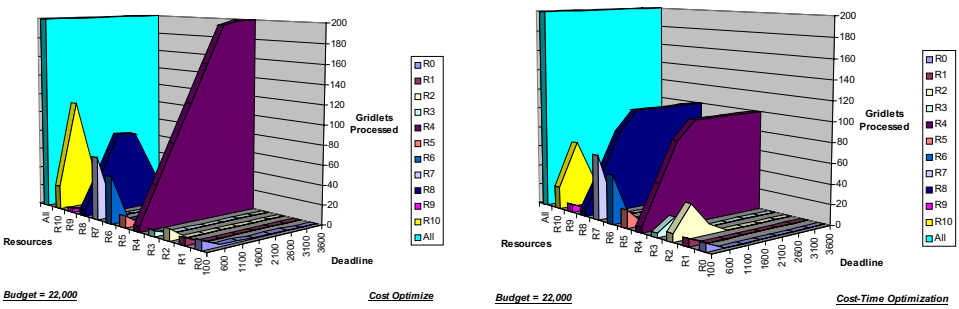
Figure 5. The number of Gridlets processed and resources selected for different budget values with a long deadline value when scheduled using the cost- and cost–time-optimization algorithms: (a) cost optimization with a short deadline; (b) cost–time optimization and a short deadline; (c) cost optimization with a long deadline; (d) cost–time optimization with a long deadline.

6. EMPIRICAL EVALUATION—SCHEDULING ON THE WORLD-WIDE GRID TESTBED

The aim of this empirical study is to demonstrate that it is feasible to implement and deploy the cost–time-optimization algorithm for scheduling applications on real Grid resources. The list of resources[‡] used in application scheduling experiments drawn from the SC2002 Global Grid Testbed

[‡]The resources used in this empirical study are different from those simulated (reference) resources as they were either inaccessible due to middleware incompatibility problems (e.g. Globus Monitoring and Discovery System software is not backwards compatible), or, in some cases, we lost access due to change of access policies.

Copyright © 2005 John Wiley & Sons, Ltd.



(a) (b)

Figure 6. The number of Gridlets processed and resources selected for different deadline values with a high budget when scheduled using the cost- and cost–time-optimization algorithms: (a) resource selection in cost optimization scheduling when the budget is high; (b) resource selection in cost–time optimization scheduling when the budget is high.

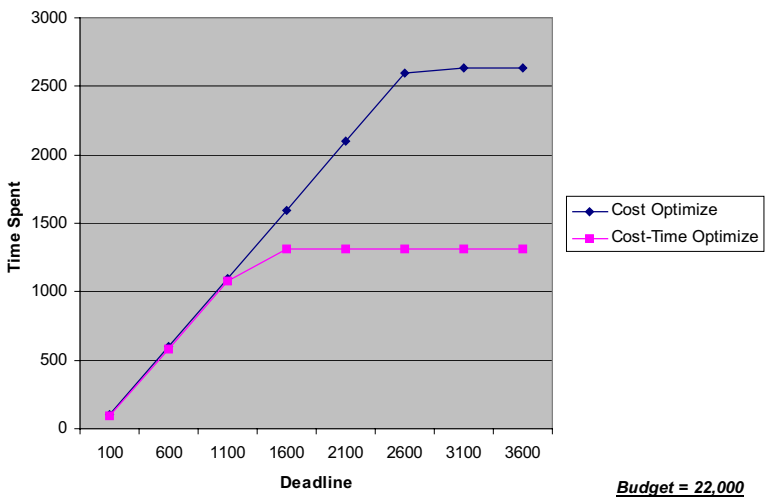


Figure 7. The time spent for processing application jobs for different deadline constraints with a large budget when scheduled using the cost- and cost–time-optimization algorithms.

Copyright © 2005 John Wiley & Sons, Ltd.

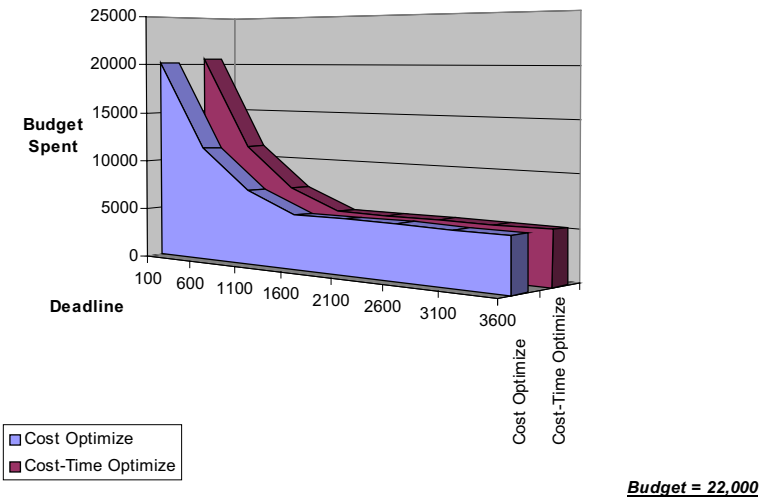


Figure 8. The budget spent for processing application jobs for different deadline constraints with a large budget when scheduled using the cost- and cost–time-optimization algorithms.

Collaboration [39] and those of the WWG [3] is given in Table II. As these distributed resources are shared among various users and no single user has control over allocation of resources, it is *impossible* to carry out *repeatable* and *comparable* evaluations as the availability of resources varies with time and there is no centralized control to create a stable environment. This situation has been noted in many early works [11,16,35] and also in empirical results presented in the rest of the section.

The cost–time-optimization strategy has been *recently* implemented through the Gridbus [40] scheduler, which was developed as a plug-in scheduler to the Nimrod-G broker using its APIs. The API implements a string-based protocol which allows a program to steer a computation through Nimrod, but in its own way rather than use the Nimrod scheduling algorithms. This way of testing experimental optimization strategies does not require Nimrod-G itself to be changed. Here, Nimrod-G performs resource discovery, selection using Gridbus schedule, and dispatching of jobs to a remote resource, starting and managing the execution of jobs and gathering the results back at the home node.

The costs shown in the Table II for each node were assigned artificially for this experiment only. However, the scheduler would find out the cost of each node from the Grid Market Directory [41] based on the GRACE trading protocols (commodity market models). Secure and remote access to all these resources is enabled through Globus middleware.

An experiment consisted of scheduling a parameter-sweep application for execution on the various nodes in our Grid testbed using Nimrod-G. The application used here is *calc*, a program that calculates mathematical functions based on the values of two input parameters. The first parameter, *length*, is an input to a mathematical function and the second parameter, *time_base_value*, indicates the expected calculation complexity in minutes plus 0–60 minutes. A plan file modeling this application

Copyright © 2005 John Wiley & Sons, Ltd.

Table II. List of Grid resources used in the experimentation.

Organization	Node details (architecture, no. of nodes, hostname)	Cost (G$ per CPU sec)
N*Grid Project Korea	Linux Cluster, 24 nodes node1001.gridcenter.or.kr	3
Vrije Universiteit, Netherlands	Linux Cluster, 144 nodes (32 available),fs0.das2.cs.vu.nl	2
N*Grid Project Korea	Linux Cluster, 16 nodes, node2001.gridcenter.or.kr	1
IIT, NRC, Canada	IBM SP, 4 nodes, hpc76.ai.iit.nrc.ca	1
Department of Physics, University of Melbourne, Australia	Linux Cluster, 10 nodes, lem.ph.unimelb.edu.au	1
Cambridge University, U.K.	Linux Cluster, 20 nodes, herschel.amtp.cam.ac.uk	1
MARCC, University of Melbourne, Australia	Linux Cluster, 8 nodes, gnet01.hpc.unimelb.edu.au	1

```
#Parameter definition
parameter length integer range from 1 to 200 step 1;
parameter time_base_alue integer default 10;
#Task definition
task definition
task main
        #Copy necessary executables depending on node type
        copy clac.SOS node:calc
        #Execute program with parameter values on remote node
        node:execute ./calc $length $time_base_value
        #Copy results file to use home node with jobname as extension
        copy node:output ./output.$jobname
endtask
```

Figure 9. Plan file for executing calc application using Nimrod-G parameter-specification language.

as a parameter sweep application using the Nimrod-G parameter-specification language is shown in Figure 9. The first part defines parameters and the second part defines the task that is to be performed for each job. As the parameter *length* varies from values 1 to 200 in steps of 1, this plan file would create 200 jobs with input values from 1 to 200. To execute each job on a Grid resource, Nimrod-G copies the program executable to a Grid node, executes the program and finally copies the results back to the user home node and stores results in the output file with jobs number as file extension.

The experiments were carried out on 8 December 2002 between 10 a.m. and 1 p.m. Australian Eastern Daylight Time. The cost- and cost–time-optimization strategies were tried out and compared.

Copyright © 2005 John Wiley & Sons, Ltd.

Table III. Summary of experiment statistics.

Scheduling strategy	Start time	Completion time	Execution time (min)	Budget consumed (G$)
Cost	10:00 a.m.	11:27 a.m.	87	188
Cost–time	11:40 a.m.	12:08 p.m.	28	277

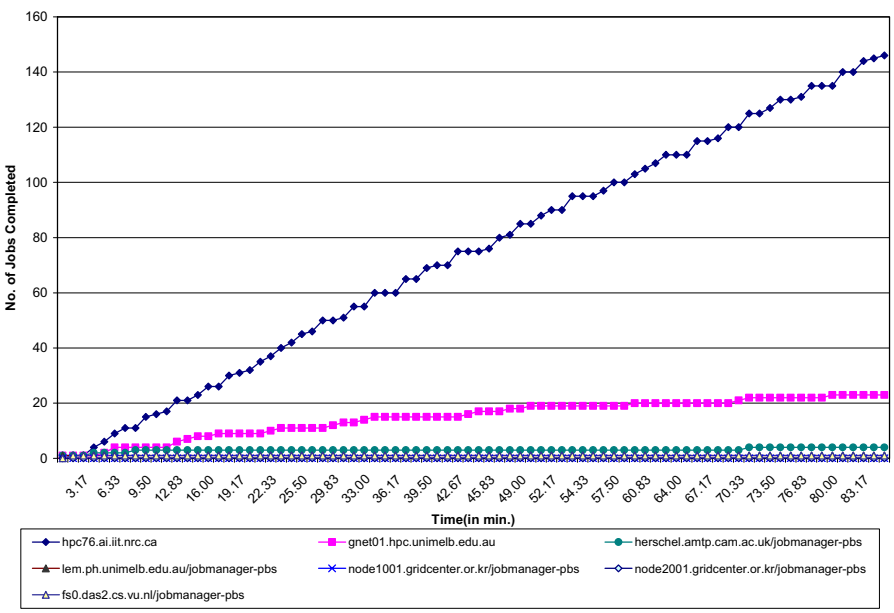


Figure 10. Cumulative graph of no. of jobs completed versus time for cost-optimization strategy.

All experiments were started with: deadline = 2 hours, budget = 600 G$. The summary of the results of these experiments is given in Table III.

The graph in Figure 10 shows cumulatively the number of jobs done against time taken. Here, we can see that the scheduler has allocated most of the jobs to the machine with the least cost per CPU cycle (i.e. hpc76.iit.nrc.ca). It does allocate some jobs to another machine (gnet01.hpc.unimelb.edu.au) so that the deadline can be achieved. Based on the performance of the first node, the scheduler realizes that it can finish the jobs within the deadline. Hence most of the jobs are allocated to the Canadian IBM-SP. Thus we see that given a fairly relaxed deadline, as in this case, the scheduler tries to execute the jobs in the least expensive way using the cost-optimization strategy.

Copyright © 2005 John Wiley & Sons, Ltd.

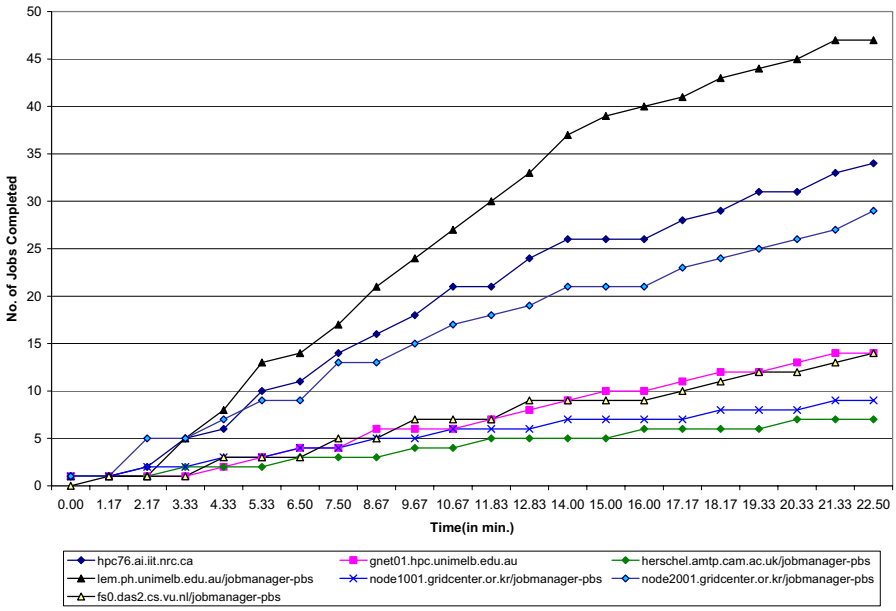


Figure 11. Cumulative graph of no. of jobs completed versus time for cost–cost-optimization.

The cumulative graph for cost–time optimization in Figure 11 shows a different case. Here, most of the jobs are executed by the three least expensive machines (lem.ph.unimelb.edu.au, node2001.gridcenter.or.kr, hpc76.ai.nrc.ca). It can be seen that, as predicted by the simulation, the scheduler performs time optimization among the machines with the least costs, and hence the University of Melbourne Physics cluster is allocated the maximum number of jobs as it is the fastest among the cheapest. However, the scheduler does allocate some jobs in the beginning to other, more expensive machines as initially two of the cheaper machines (lem.ph.unimelb.edu.au and hpc76.ai.iit.nrc.ca) are slow to pick up jobs.

In terms of the budget spent during execution, there is a deviation between the results of simulation and the actual experimental results. Simulation predicts that with a relaxed deadline, the budget spent should nearly be the same for both the cost–time- and cost-optimization algorithms. However, in these experiments, the cost–time-optimization method is found to be more expensive than the cost-optimization method (see Table III). This is due to a huge variation in the availability of some of the Grid resources during the cost–time-optimization scheduling. For example, the Canadian IBM SP machine was able to process many jobs during the cost-optimization scheduling experiment, but the available processing capability had reduced during the cost–time-optimization. This impacted on the amount of budget that was being spent. If the least expensive machines had performed well in the beginning, the cost would have remained the same.

Copyright © 2005 John Wiley & Sons, Ltd.

7. SUMMARY AND CONCLUSION

Computational Grids enable the sharing, discovery, selection, and aggregation of geographically distributed heterogeneous resources for solving large-scale applications. We proposed computational economy as a metaphor for managing the complexity that is present in the management of distributed resources and their allocation. It allows allocation of resources depending on the users' QoS requirements such as the deadline, budget, and optimization strategy. This paper proposed a new deadline- and budget-constrained scheduling algorithm called *cost–time optimization*. We developed a scheduling simulator using the GridSim toolkit and evaluated the new scheduling algorithm by comparing its performance and quality of service delivery with the cost-optimization algorithm.

When there are multiple resources with the same cost and capability, the cost–time-optimization algorithm schedules jobs on them using the time-optimization strategy for the deadline period. From the results of scheduling experiments for many scenarios with a different combination of deadline and budget constraints, we observe that applications scheduled using the *cost–time*-optimization algorithm are able to complete earlier than those scheduled using the cost-optimization algorithm, without incurring any extra expenses. This establishes the superiority of the new deadline- and budget-constrained cost–time-optimization algorithm in scheduling jobs on global Grids. The cost–time-optimization strategy has also been implemented in the Gridbus scheduler to demonstrate that it is feasible to implement and deploy the cost–time-optimization algorithm for scheduling applications on global Grids.

SOFTWARE AVAILABILITY

The GridSim toolkit and the economic Grid broker simulator with source code can be downloaded from: http://www.gridbus.org/gridsim/.

The Nimrod-G broker can be downloaded from: http://www.csse.monash.edu.au/∼davida/nimrod/.

The Gridbus broker can be downloaded from: http://www.gridbus.org/broker/.

ACKNOWLEDGEMENTS

We thank Rob Gray (DSTC) for his comments on improving the paper and Marcelo Pasin (Federal University of Santa Maria, Brazil) for his help in modeling resources with SPEC benchmark ratings. We would like to thank participants in the SC2002 Global Grid Testbed Collaboration for providing us with access to the resources that have made large-scale experiments possible.

We are grateful to the reviewers for their meticulous and constructive comments that helped us in improving the quality of this paper substantially.

REFERENCES

1. Foster I, Kesselman C (eds.). *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers: San Francisco, CA, 1999.
2. Oram A. (ed.). *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly Press, 2001.
3. Buyya R. Economic-based distributed resource management and scheduling for Grid computing. *PhD Thesis*, Monash University, Melbourne, 2002.

Copyright © 2005 John Wiley & Sons, Ltd.

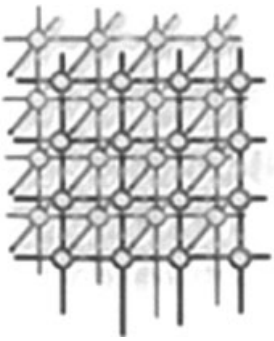
4. Abramson D, Giddy J, Kotler L. High performance parametric modeling with Nimrod/G: Killer application for the global Grid? *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2000)*. Cancun, Mexico, 1–5 May 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000.
5. Buyya R, Abramson D, Giddy J. Nimrod/G: An architecture for a resource management and scheduling system in a global computational Grid. *Proceedings of the 4th International Conference and Exhibition on High Performance Computing in Asia-Pacific Region (HPC ASIA 2000)*, Beijing, China, May 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000.
6. Buyya R, Abramson D, Giddy J. An economy driven resource management architecture for global computational power Grids. *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, Las Vegas, June 2000. CSREA Press, 2000.
7. Buyya R, Giddy J, Abramson D. An evaluation of economy-based resource trading and scheduling on computational power Grids for parameter sweep applications. *Proceedings of the 2nd International Workshop on Active Middleware Services (AMS 2000)*, Pittsburgh, PA, August 2000. Kluwer, 2000.
8. Casavant TL, Kuhl JG. A taxonomy of scheduling in general purpose distributed computing. *IEEE Transactions on Software Engineering* 1988; **14**(2).
9. Feitelson DG, Rudolph L (eds.). *Proceedings of the 5th IPPS/SPDP'99 Workshop Job Scheduling Strategies for Parallel Processing (JSSPP 1999)*, San Juan, Puerto Rico, April 1999 (*Lecture Notes in Computer Science*, vol. 1659). Springer: Heidelberg, 1999.
10. Buyya R, Branson K, Giddy J, Abramson D. The virtual laboratory: Enabling molecular modelling for drug design on the World Wide Grid. *Technical Report CSSE-103*, Monash University, 2001.
11. Buyya R, Murshed M. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *Concurrency and Computation: Practice and Experience* 2002; **14**(13–15).
12. Litzkow M, Livny M, Mutka M. Condor—a hunter of idle workstations. *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988. Editrice Compositori/Press: Bologna, Italy, 1988.
13. Frey J, Tannenbaum T, Foster I, Livny M, Tuecke S. Condor-G: A computation management agent for multi-institutional Grids. *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, San Francisco, CA, 7–9 August 2001.
14. Czajkowski K, Foster I, Karonis N, Kesselman C, Martin S, Smith W, Tuecke S. A resource management architecture for metacomputing systems. *Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1998.
15. Weissman J, Grimshaw A. A federated model for scheduling in wide-area systems. *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC)*. Sage Publications: Thousand Oaks, CA, 1996.
16. Berman F, Wolski R. The AppLeS project: A status report. *Proceedings of the 8th NEC Research Symposium*, Germany, May 1997. Elsevier Press: Amsterdam, The Netherlands, 1997.
17. Casanova H, Kim M, Plank J, Dongarra J. Adaptive scheduling for task farming with Grid middleware. *The International Journal of High Performance Computing* 1999; **13**(3).
18. Hawick K *et al.* DISCWorld: An environment for service-based metacomputing. *Future Generation Computing Systems* 1999; **15**.
19. Chun B, Culler D. User-centric performance analysis of market-based cluster batch schedulers. *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, Berlin, Germany, May 2002. CERN: Geneva, Switzerland, 2002.
20. Waldspurger C, Hogg T, Huberman B, Kephart J, Stornetta W. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, February 1992.
21. Hacker T, Thigpen W. Accounting models research group. *Global Grid Forum*. http://www.gridforum.org/5_ARCH/ACCT.htm.
22. Wolski R, Plank J, Brevik J, Bryan T. Analyzing market-based resource allocation strategies for the computational Grid. *International Journal of High-performance Computing Applications* 2001; **15**(3).
23. Casanova H, Obertelli G, Berman F, Wolski R. The AppLeS parameter sweep template: User-level middleware for the Grid. *Proceedings of the IEEE SC 2000: International Conference Networking and Computing*, Dallas, TX, November 2000. TurboWorx: Shelton, CT, 2000.
24. Sato M, Nakada H, Sekiguchi S, Matsuoka S, Nagashima U, Takagi H. Ninf: A network based information library for global world-wide computing infrastructure. *Proceedings of the International Conference on High Performance Computing and Networking Europe (HPCN Europe)*, Vienna, Austria, 28–30 April 1997. IEEE Computer Society Press: Los Alamitos, CA, 1997.
25. Haritsa J, Carey MJ, Livny M. Earliest deadline scheduling for real-time database systems. *Proceedings of the Real-Time Systems Symposium*, 1991.
26. Biyabani S, Stankovic J, Ramamritham K. The integration of deadline and criticalness in hard real-time scheduling. *Proceedings of the Real-Time Systems Symposium*, 6–8 December 1998.

Copyright © 2005 John Wiley & Sons, Ltd.

27. Aydin H, Mejia-Alvarez P, Melhem R, Mosse D. Optimal reward-based scheduling of periodic real-time tasks. *Proceedings of the IEEE Real-Time Systems Symposium*, Phoenix, AZ, December 1999.
28. Sullivan WT III, Werthimer D, Bowyer S, Cobb J, Gedye D, Anderson D. A new major SETI project based on Project Serendip data and 100,000 personal computers. *Proceedings of the Fifth International Conference on Bioastronomy*, 1997. Available at: http://setiathome.ssl.berkeley.edu/.
29. Pande V. Protein folding on Internet-wide distributed computing (Folding@Home project). http://www.stanford.edu/group/pandegroup/Cosm/ [2001].
30. Blackstone. Post-genomic challenges drive life sciences to high throughput computing (HTC), *White paper*, April 2001. Available at: http://www.blackstonecomputing.com/products/whitePapers/.
31. CERN. The Large Hadron Collider (LHC) Computing Grid Project for high energy physics data analysis. http://lcg.web.cern.ch/LCG/ [May 2003].
32. Abramson D, Power K, Sosic R. Simulating computer networks using clusters of PCs. *HPC-TelePar'99 at the 1999 Advanced Simulation Technologies Conference (ASTC'99)*, San Diego, CA, 11–15 April 1999.
33. Smallen S, Casanova H, Berman F. Applying scheduling and tuning to on-line parallel tomography. *Proceedings of the IEEE/ACM Supercomputing (SC 2001) Conference*, Denver, CO, November 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001.
34. Casanova H, Bartol T, Stiles J, Berman F. Distributing MCell simulations on the Grid. *The International Journal of High Performance Computing and Supercomputing Applications* 2001; **15**(3).
35. Casanova H. SimGrid: A toolkit for the simulation of application scheduling. *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, Brisbane, May 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001.
36. Song H, Liu X, Jakobsen D, Bhagwan R, Zhang X, Taura K, Chien A. The MicroGrid: A scientific tool for modeling computational Grids. *Proceedings of Supercomputing (SC 2000) Conference*, Dallas, TX, November 2000. IEEE Press: Piscataway, NJ, 2000.
37. SPEC. SPEC CPU2000 results. http://www.specbench.org/osg/cpu2000/results/cpu2000.html [30 January 2002].
38. Buyya R, Abramson D, Giddy J, Stockinger H. Economic models for resource management and scheduling in Grid computing. *Concurrency and Computation: Practice and Experience* 2002; **14**(13–15).
39. SC2002 Global Grid Testbed Collaboration. http://scb.ics.muni.cz/static/SC2002/.
40. Buyya R, Venugopal S. The Gridbus toolkit for service oriented grid and utility computing: An overview and status report. *Proceedings of the First IEEE International Workshop on Grid Economics and Business Models (GECON 2004)*, Seoul, Korea, 23 April 2004. IEEE Press: New Jersey, 2004; 19–36.
41. Yu J, Venugopal S, Buyya R. A market-oriented Grid directory service for publication and discovery of Grid service providers and their services. *Technical Report*, Department of Computer Science and Software Engineering, University of Melbourne, 2002.
42. Buyya R. The World-Wide Grid. http://www.gridbus.org/wwg/.

Copyright © 2005 John Wiley & Sons, Ltd.

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2008)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1353



Fault-tolerant execution of large parameter sweep applications across multiple VOs with storage constraints

Shahaan Ayyub*,†, David Abramson, Colin Enticott, Slavisa Garic and Jefferson Tan

Faculty of Information Technology, Monash University, Melbourne, Vic. 3145, Australia

SUMMARY

Applications that span multiple virtual organizations (VOs) are of great interest to the e-science community. However, our recent attempts to execute large-scale parameter sweep applications (PSAs) for real-world climate studies with the Nimrod/G tool have exposed problems in the areas of fault tolerance, data storage and trust management. In response, we have implemented a task-splitting approach that facilitates breaking up large PSAs into a sequence of dependent subtasks, improving fault tolerance; provides a garbage collection technique that deletes unnecessary data; and employs a trust delegation technique that facilitates flexible third party data transfers across different VOs. Copyright © 2008 John Wiley & Sons, Ltd.

Received 11 December 2007; Revised 30 March 2008; Accepted 7 May 2008

KEY WORDS: e-science; parameter sweep applications; Grid

1. INTRODUCTION

The computational Grid aggregates computational power and storage capacity by coupling together distributed CPU, network and storage resources [1]. The scale and nature of Grid testbeds make it possible to solve particular challenging problems in science and engineering using *parameter sweep*

*Correspondence to: Shahaan Ayyub, Faculty of Information Technology, Monash University, Melbourne, Vic. 3145, Australia.
†E-mail: ayyub.shahaan@infotech.monash.edu.au

Contract/grant sponsor: Australian Research Council
Contract/grant sponsor: CSIRO Division of Atmospheric Research

Copyright © 2008 John Wiley & Sons, Ltd.



applications (PSAs). PSAs are generally specified as a set of experiments, each representing a *parameterized instance* of a user-defined model. In principle, each instance is executed with a distinct set of parameter combinations on a number of resources. The parameters control the behaviour of the model and allow users to explore different design scenarios or options. The case study in this paper, for example, varies environmental conditions in a simulation of burning grasslands over a period of 21 years in a particular geographical area. PSA models have been applied in different areas including physical sciences [2,3], mathematics [4] and biology [5].

Conceptually, implementing a PSA is not difficult. For example, a simple Cartesian product of a range of parameters would generate a set of experiments, which can then be executed in parallel on a number of resources. However, in practice, effective deployment of PSAs on the Grid poses intricate management issues, making the handling of the experiments error-prone and unwieldy to perform manually. As a result, parametric modelling tools such as Nimrod [1], APST [6] and GridBus [7] were developed to simplify this process. These tools allow e-scientists to concentrate on their science while hiding the complexities of deploying and managing applications on widely distributed resources.

We recently attempted to execute extremely large PSAs on the Grid, and this challenged solutions to task, data and trust management.

Task management: If an individual computation takes a long time, then executing it as one continuous run becomes error-prone and unwieldy. This is particularly true in the presence of unreliable network and resource conditions found on dynamic platforms such as the Grid. When a runtime error occurs, the computation may abort its execution and must be restarted from the very beginning. All intermediately generated data will be lost. To address this problem, applications can be split and *check-pointed* [8]. In the event of a failure, a program can resume from the most recently saved check-point, rather than restarting the entire application. Although this idea is not new, we have implemented a simple user-driven technique supporting Grid-based check-pointing in order to achieve fault tolerance and improved application performance. The check-pointing is explicit; the user can set the granularity of check-points when specifying their application.

Data management: Task-splitting and check-pointing exacerbate data management problems because it becomes necessary to store the application state for each check-point. Moreover, it might also be necessary to transfer application's state to another resource should the task be migrated there. It is desirable to simplify the specification of data-handling mechanisms as we have done, through basic primitives built into the specification syntax and semantics that we use in describing applications. On the other hand, data transfer costs must be considered carefully. Some scheduling heuristics are more effective than others because they place downstream computations on resources such that the cumulative cost of transferring the associated files and estimated computation time is minimized [6,9]. This improves the overall application performance by opportunistically migrating downstream sub-computations to better resources. However, the amount of data generated by multiple sub-computations may exceed the storage limits set on resources, which may restrict the further scheduling of jobs to those resources. To solve this problem, we built a garbage collection technique that keeps the amount of generated data within the permitted storage limits, even when multiple applications are simultaneously executed on a single resource.

Trust management: Large-scale applications require as many computational resources as possible to achieve high throughput. Normally, a single *virtual organization* (VO) [1] is formed by several entities with common goals and an agreement to share computational and data resources. However,

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

one VO alone may not provide sufficient resources for particularly large projects; hence, it may be necessary for a Grid testbed to span multiple VOs. Typically, Grid resources belonging to a particular VO only trust users authorized by the *Certificate Authority* (CA) of that VO. As a result, data transfers across resources, often referred to as *third party* transfers, may not be possible because those resources belong to different VOs. We encountered such difficulties in the Australian climate studies discussed throughout this paper. To address this, we have implemented a trust delegation model that facilitates flexible third party data transfers across different VOs.

The devised techniques are implemented in Nimrod/G, a tool that executes Grid-based PSAs [1]. Nimrod/G is a parameter sweep middleware that includes the ability to describe PSAs and launch them across a Grid testbed using a meta-scheduler. Apart from the ability to launch parameterized instances as jobs across the testbed, Nimrod/G is able to manage their execution dynamically with the ability to respond to errors on the network or by individual nodes on the testbed. As a result, Nimrod/G gracefully migrates the downstream sub-computations to better resources when a job or resource fails or when there is resource contention. In addition, Nimrod/G efficiently co-schedules multiple computations on a single resource, while keeping the cumulative data generated by these computations within permitted storage limits. Furthermore, results show that, while migrating sub-computations to resources of different VOs, Nimrod/G appropriately copies the state files associated with every sub-computation.

The next section describes the challenges encountered while executing large PSAs on a multi-VO Grid. Section 3 describes our implemented techniques and the resulting improvements to the application model. Section 4 illustrates the efficacy of the implementation through experimental analysis. In Section 5, we summarize the lessons that we have learned during our experience with the real-world climate studies related in this paper. In Section 6, we briefly present related works. Section 7 concludes the paper and mentions some of our future endeavours.

2. CHALLENGES

The experiment described in this paper studies the effects of the burning of the Savannah on the climate of Northern Australia [10], using the Conformal-Cubic Atmospheric Model (CCAM) [11]. The inputs to CCAM, which include a set of files, can be tuned to simulate different climatic scenarios of a particular geographic location. For example, one of the parameters, the time parameter, specifies the duration for which the analysis should be performed on a particular geographical location. We generated an order of 90 instances of CCAM for the experiment [12]. We describe the challenges encountered in task, data and trust management below.

2.1. Task management

The application was initially set up as shown in Figure 1. The same application executes as 90 separate instances, each with a distinct set of parameter values stored within distinct files. Each instance takes such a file as input, as well as some common data files that do not vary, computes a scenario and then generates an output for further investigation. Each computation modelled 21 years of weather conditions requiring 6–7 weeks of CPU time to complete. Nimrod/G harnesses several computational nodes in order to execute these instances in parallel as distinct jobs. However, as

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

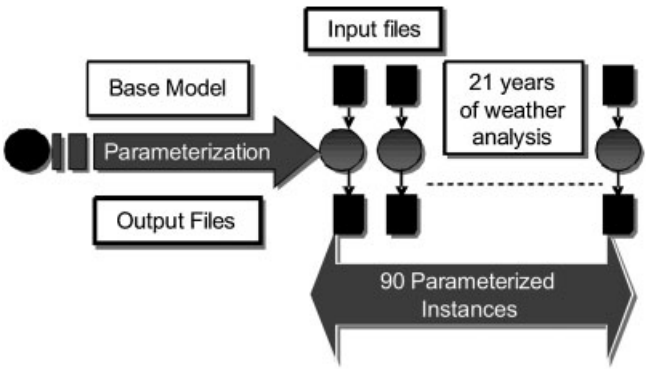


Figure 1. Old application model.

mentioned earlier, executing such long running jobs as a single, continuous run is prone to failures. Apart from possible I/O or processing failures, network failures will also disrupt the execution of those jobs. Although Nimrod/G is able to restart failed jobs, or start them on another node if the previous node had failed, restarting a long running computation (6–7 weeks in our case) due to a job or resource failure would waste huge amounts of time already invested in the failed run. Additional time is also incurred to re-run the computation from the very beginning, particularly when the job requires pre-processing or if it must be submitted to a new computational resource instead. We therefore addressed this issue with long duration jobs by *task-splitting*, structuring each as a group of 252 subtasks in a sequential workflow. The CCAM application must process data from 252 months in sequence, whereas each month's data input data can be treated independent of others. Each month, therefore, was used as a *time step* within each of the 90 workflows. The challenge is therefore twofold, in terms of application design and check-point process. The CCAM application was neither originally designed for this purpose nor did the PSA specification include sequential workflow features to begin with. At the same time, while the notion of check-pointing is well known, the fact that we are check-pointing subtasks within a sequential workflow requires special consideration of the dependencies among subtasks.

2.2. Data management

Task-splitting, as described earlier, generally adds to the data management problem. Each subtask must generate persistent output that will be taken as input by the next subtask in the sequential workflow. In comparison, if we do not exercise task-splitting, then each long-duration job will handle data internally, e.g. as data in the memory allocated to the running job. This is obviously simpler and, in fact, quicker in the absence of file I/O. On the other hand, appropriate task allocation techniques can improve an application's performance by migrating the downstream computations to resources with better CPU performance and network connectivity. This provides the additional advantage of dynamic and adaptive resource allocation in response to drastic changes in resource conditions across the Grid testbed, such that subtasks can be initiated or migrated across to better resources. On the other hand, these subtasks must have access to the data they require. Although

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

convenient file transfer mechanisms are available to Nimrod/G, user applications can easily conflict with disk quotas. Existing heuristics assume unlimited disk space on Grid resources [13,14], which is unrealistic on Grid testbeds where users are typically allocated limited disk space. In our test case, each subtask generated about 70 MB of output. Combined with the common input files of around 250 MB in size, this exceeded the storage available on some testbed resources. On the other hand, much of the output data are only required temporarily as the output of one subtask is required only as input to the next sequential task. Therefore, it is not a straightforward case of requiring a more generous disk quota, something that is not always possible to begin with. What is required is an appropriate data management technique that will properly delete files that become obsolete as the Grid workflow progresses.

2.3. Trust management

To deal with the scale of our application, we executed our experiments on the Pacific Rim Applications and Grid Middleware Assembly or PRAGMA Grid testbed [10]. The PRAGMA testbed consists of a large set of resources spanning across several VOs. The Globus toolkit [1] was used to build the basic infrastructure for providing resource access and remote data transfer services.

Figure 2 shows a sample of such Grid testbed models integrated from several individual VOs. Table I lists the exclusive CA domains that we involved in the testbed. Typically, all resources in an individual VO trust a single CA [1], and thus all resources can communicate freely and authenticate data transfers against this single CA. This is important when intermediate files are transferred directly from one resource to another as part of a data-dependent computation within the PSA. However, if the testbed spans multiple VOs as discussed, then a sending resource and a receiving resource may trust different CAs, making third party transfers difficult. One solution to this problem is to route the data through a resource trusted by both parties. However, this incurs additional communication overhead, as the data now need to be transferred through an intermediary resource. To address this, we have designed and implemented a new mechanism that delegates trust across multiple VOs, allowing the direct transfer of data between resources across different VOs despite having mutually exclusive CA domains.

A relevant but separate challenge is the issue of internetworking restrictions. Single VOs involve independent real-world organizations, for which the underlying internetworking is typically built on top of the Internet. The risky nature of the Internet provides the motivation for using firewalls to protect the networks within these organizations. The restrictions caused by firewalls sometimes result in the failure of Grid transactions. Note that firewall issues are distinct from trust issues addressed by CA domain infrastructures. In the latter context, trust domains are represented and addressed within VOs, with authorization based on certificates signed by the CA [1]. Firewall issues reflect the autonomy of *real-world* organizations, not virtual ones. Firewall restrictions are not based on certificates, therefore, but at lower levels of the network infrastructure, typically at the transport (TCP or UDP ports) or internet levels (IP addresses). We address firewall issues in a separate project, *Remus*, which reroutes connections through tunnels and proxies using the most appropriate transport mechanism for a given firewall situation [15]. However, these issues are beyond the scope of this paper, as we did not encounter them in the testbed described here. Despite the involvement of different institutions across the U.S., Japan, Taiwan, Thailand and Australia, we did not have to deal with firewalls at all.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

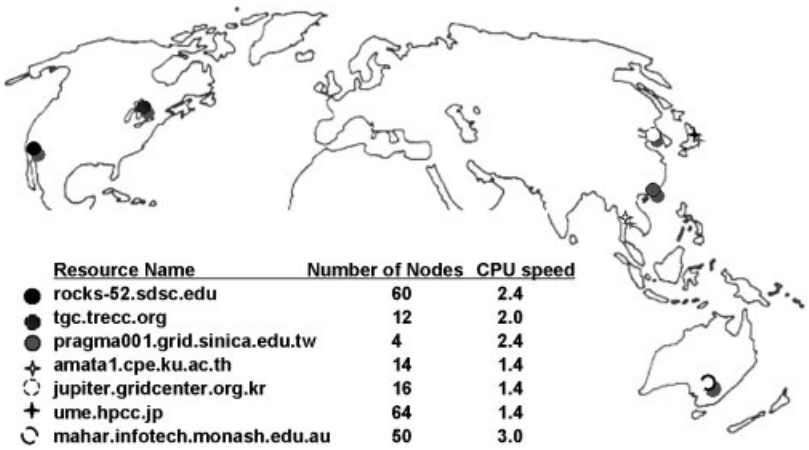


Figure 2. Grid testbed model.

Table I. List of CA domains involved in the climate study described in this paper.

CA domain	Region
Australian Partnership for Advanced Computing (APAC)	Australia
National Partnership for Advanced Computational Infrastructure (NPACI)	U.S.
National Institute of Advanced Industrial Science and Technology (AIST)	Japan
Korea Institute of Science and Technology Information (KISTI)	Korea
Academia Sinica (AS)	Taiwan
National Center for Supercomputing Applications	U.S.

3. SOLUTIONS

In this section, we present the solutions that we came up with in order to meet the challenges described previously.

3.1. Task management

Many scientific applications, such as CCAM, provide explicit check-pointing at the application level. In our case, we split large simulations, each computing 21 years of weather scenarios, into a sequence of 252[‡] dependent monthly sub-computations, linked through *state files* containing check-pointed data (Figure 3). A state file is an intermediary file containing the state information from one subtask that is to be used by its subsequent child. As stated earlier, challenges in task management are in modelling the task-splitting as well as in handling the dependencies across subtasks.

[‡]21 yearly computations $*$12 months $=$252 monthly computations.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

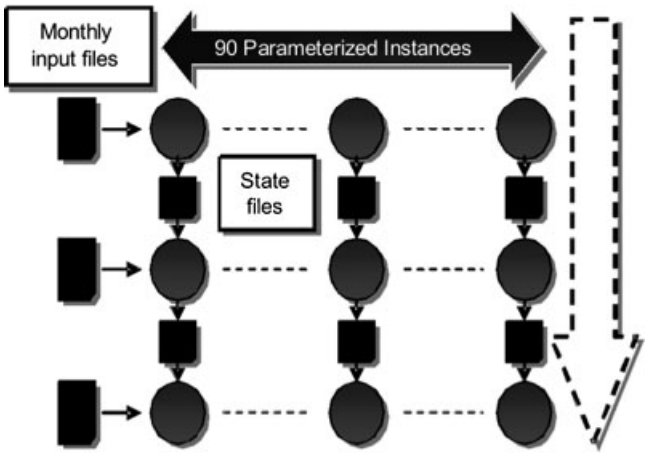


Figure 3. New application model with sequential dependencies.

It is difficult to express task-splitting as described in this paper succinctly using existing parametric modelling languages embedded in existing tools such as GridBus [7], ICENI [16] and earlier versions of Nimrod/G [1]. Other declarative languages, such as APST [17], use comprehensive but large XML-based descriptions to specify experiments [18].

Nimrod/G uses a simple declarative language, called *planfile*, to specify parametric modelling experiments. Figure 4 shows the syntax of the current implementation of what we refer to as a *planfile*. Normally, Nimrod/G takes the Cartesian product of the declared parameters and creates a parametric hyperspace of Cartesian points[§], each representing a single task. To specify check-point dependencies in parametric spaces, we introduce the *seqameter* (Figure 4, line 3.3), i.e. a *sequential* parameter. A seqameter behaves like a normal parameter, but its inclusion breaks a parametric instance into a set of temporally dependent sub-instances (see Figure 3). In our case study, the values in the seqameter range distinguish one subtask from another within a set of subtasks that must execute in sequence. In comparison, the values in the parameter space distinguish one independent parametric instance from another, making them distinct as they may execute in parallel. We have integrated seqameters in Nimrod/G's planfile syntax, which provide explicit control to the user in setting the granularity of the check-points.

As seen in Figure 3, instead of having 90 jobs for 90 parametric instances, we instead model the PSA as 90 sequences of subtasks. Each subtask receives monthly input files as well as initial data coming from the subtask above it, if any. Figure 4 describes the PSA and how seqameters are used. The 'expnr' integer parameter distinguishes each of the 90 parametric instances. The 'timestep' seqameter distinguishes, for each parametric instance, each of the 252 check-point steps: one for each month during the simulation period of 21 years. The various execute commands in the planfile use the parameter and seqameter values as references to the input and state files to use for the next subtask to execute. Note that the splitting process introduced dependencies among the subtasks.

[§] A Cartesian point can be defined as (a, b) where $a \in A$ and $b \in B$, A and B being the two parametric sets.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



```
1.  parameter expnr integer range from 1 to 90 step 1;
2.  seqameter timestep integer range from 1 to 252 step 1;
3.  task main
    3.1. inputs cms.${expnr}.`/bin/echo $[${timestep} 1]`.tar.gz
         vf_ccam.$timestep
    3.2. output cms.${expnr}.${timestep}.tar.gz on create f1
    3.3. sequemeters timestep
    3.4. parameters expnr
    3.5. node:execute tar xfz cms.${expnr}.*.tar.gz
    3.6. node:execute /bin/mv vf_ccam.$timestep
         cms.$expnr/input/vf_ccam.$timestep
    3.7. node:execute cd cms.${expnr} && /cms_prep.ksh
         ${expnr}${timestep} >> ../../output 2>&1 && cd..
    3.8. node:execute cd cms.${expnr}/output && .
         ${HOME}/bin/globpea >> ../../output 2>&1 && cd ../../
    3.9. node:execute cd cms.${expnr} && /cms_postp.ksh
         ${expnr}${timestep} >> ../../output 2>&1 && cd..
    3.10.node:execute tar cfz cms.${expnr}.${timestep}.tar.gz
         cms.${expnr}
4.  endtask
5.  task f1
    5.1. parameters None
    5.2. seqameters None
    5.3. copy node:$FILE local,root:
    5.4. node:execute $HOME/Sreplicate s brecca -2.vpac.org Obj1
6.  endtask
```

Figure 4. New planfile syntax.

As the existing heuristics [19] in Nimrod/G were incapable of handling such dependencies, new scheduling heuristics were implemented to schedule tasks to resources based on

- resource performance and network conditions;
- exploiting the parallelism among the application components and
- respecting both task precedence and data affinity.

We implemented some existing algorithms, namely Sufferage, XSufferage [6] and a LoadBalance heuristic, in the Nimrod/G scheduler. Each user-selected heuristic attempts to optimize task allocation with a specific objective function. Sufferage allocates a node to a task that would 'suffer' most if it is not allocated to that node. XSufferage is the cluster-level implementation of Sufferage. LoadBalance distributes equal computational workload across testbed nodes with homogeneous CPU configuration and network connectivity. Sufferage and XSufferage require information about the network links and computational resource performance, which can be obtained from resource monitoring tools such as the *network weather service* (NWS) [20] and Ganglia [21]. Apart from this information, statistical logs about the past performance of the resources can also be used in order to optimize task allocation. Nimrod/G uses both NWS traces and statistical logs to optimize task scheduling. It is therefore necessary to assume that the Grid resources support both NWS and other forms of statistical logging.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

3.2. Data management

As mentioned earlier, Nimrod/G provides mechanisms for PSAs to conveniently transfer files across the testbed, e.g. the 'copy' command. In addition, the planfile syntax allows for the invocation of command line operations or the execution of system binaries. In Figure 4, apart from being able to execute shell scripts included in the application, the plan file may also cause the execution of Unix utilities such as 'tar' for archiving or unarchiving files. However, users are occasionally interested in performing operations on the state files, for example, backup instructions to store the file at a particular storage resource broker [22] location. Nimrod/G can be instructed to do this using the new planfile syntax. A specialized Nimrod task can be associated with these files, which can then be executed when a specific event occurs, such as file generation or copying (Figure 4, line 3.2). This function is declared in the plan file (Figure 4, lines 5–6) as a normal task but it consists of some extra processing instructions.

If specialized operations are not associated with the state files, then those files are removed once the subtask requiring them finishes its execution. Similarly, all other files that are not required by any other task as an input could be removed from the resources to free up some space. We therefore implemented garbage collection to remove all such files to comply with storage restrictions. This allowed unrestricted scheduling of multiple jobs on the same resource without the scheduler having to bother with space constraints. Collecting garbage in this manner solved data management problems to a large extent.

3.3. Trust management

Coupling resources belonging to multiple VOs together provides a larger resource pool than what would be available in a single VO. However, in multi-VO Grid testbeds, the scheduler might allocate a task to a resource within a VO that is different from that of its parent. As the parent's state file must be passed on to its child subtask, a third party transfer across resources in different VOs will become necessary. As described earlier, third party data transfers across VOs cannot proceed unless trust conflicts are first resolved. In principle, if the user is authorized to access resources from two different VOs, then it should be possible to perform third party data transfers between the resources. In the Globus Security Infrastructure (GSI) [1], a client must possess legitimate credentials that are validated against the CA trusted by the resource to be accessed. To access resources from different VOs, and thus, across different CAs, distinct credentials valid for each CA are required. Each valid credential is the basis for a *proxy certificate* that authenticates the user and authorizes access to resources of the associated VO.

Proxy certificates, which can be used to perform third party transfers, are available to Nimrod/G. However, Nimrod/G supports a range of different Grid middleware apart from Globus. For example, Nimrod/G currently supports file transfers across resources managed by Condor [13], PBS [23], Sun GridEngine [24], Bproc [25] and Fork. We therefore implement the third party transfer mechanism within Nimrod/G to transcend the heterogeneity among transfer methods that might otherwise be involved. Third party transfers are implemented using *Nimrod agents* and *Nimrod Remote File Servers* and secured through public key authentication. Nimrod/G manages the execution of tasks using Nimrod agents that run on the resource where a user's tasks are. Nimrod agents also manage data transfers normally required by tasks; however, third party transfers require Nimrod remote file

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



servers. To validate transfer requests from Nimrod agents involving a given resource, a remote file server is launched on it. Distinct session keys for each resource, say x and y, are generated and maintained at the user's local site in the Nimrod database. In order to perform the transfer from resource x to y, the session key required to access resource y is copied to x. The Nimrod agent on x then uses the copied key, on behalf of the user, to validate the request with the Nimrod remote file server on y. Once validated, the Nimrod remote file server facilitates the transfer operation.

In this manner, we provide a secure third party mechanism across resources from different VOs, which may support different standards. The actual transfer method relies on Nimrod/G, without having to introduce another delegation layer such as WebCom's CCA [26]. Note that the user does not need to install Nimrod/G on remote resources. Besides, Nimrod/G performs these operations by launching remote agents and file servers on distributed resources. We are thus able to use resources belonging to multiple VOs as a single 'Grid,' leveraging its distributed storage capacity.

4. EXPERIMENTS AND RESULTS

Here, we present the results of the Savannah experiment on the PRAGMA [26] testbed (Figure 5), consisting of seven resources from six different countries. Each resource had its own CA domain and showed dynamic availability patterns during the experiment.

4.1. Task management

Figure 6 shows the state transition diagrams of two sub-computations. The upper half of each state node shows the allocated resource, whereas the lower half describes the current state of the job. The sub-computations were migrated from one resource to another mainly due to

- Grid resource failures, when the resource that was allocated to the job eventually failed.
- Input file copy failures, when the file copy failed due to resource failure or network failure.
- Resource unavailability, when the front node's job queues were too long or the resource was too busy.
- Job failures, when the job failed due to missing libraries or some unexpected runtime error.
- Disk space shortage, when the input files could not be copied to the scheduled resource due to disk space limitations.

Nimrod/G is designed to migrate the jobs to other available resources to recover from these faults. Any of the faults above invariably results in the failure of Nimrod to guarantee a working environment in computational nodes. Computational nodes are remotely managed through Nimrod agents. Faults will be relayed by agents to the Nimrod scheduler or will cause agents themselves to fail and lose contact with the scheduler and other server processes. Either way, Nimrod/G reacts by presuming that the associated jobs have failed and then launching them on other nodes. Furthermore, the heuristics implemented in Nimrod/G provides a range of objective functions to optimize scheduling.

Such fault recovery and adaptability are difficult to achieve if the applications, of the scale described in this paper, were executed under the conventional parameter sweep model.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

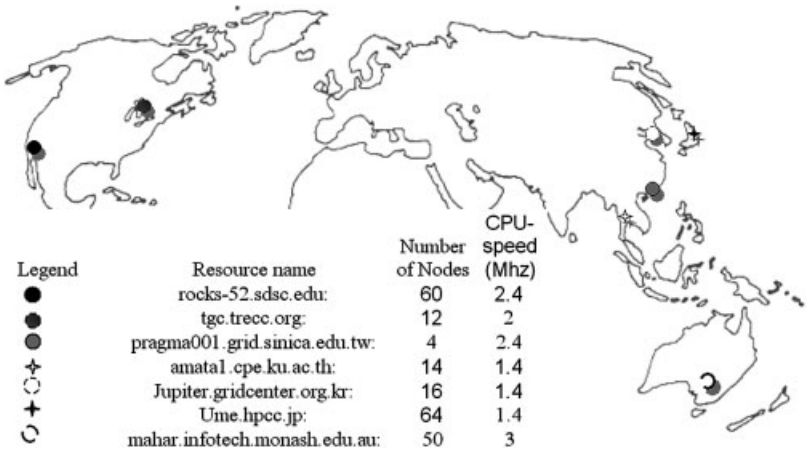


Figure 5. Testbed configuration.

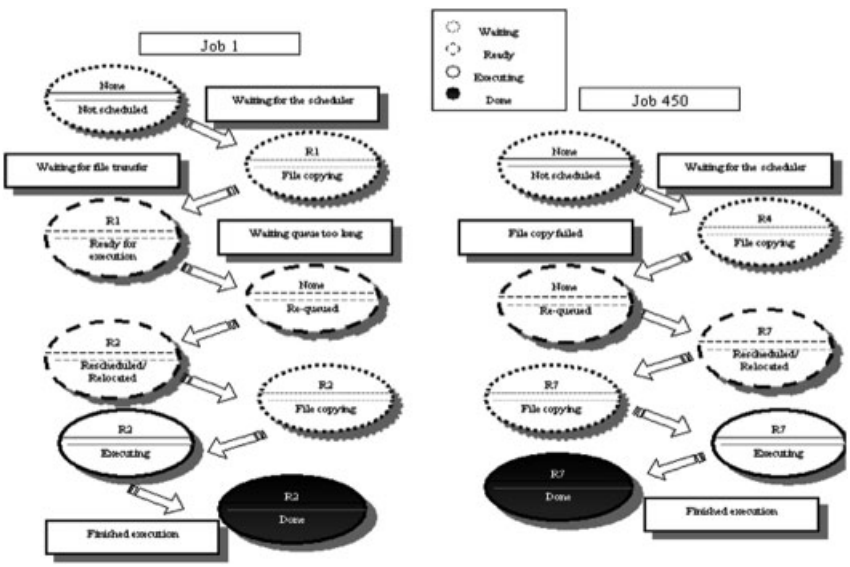


Figure 6. Diagrams of two sub-computations.

4.2. Data management

Figure 7 shows the disk space usage of the experiment running on different resources. The graph, which covers a few days of the whole experiment, illustrates dynamic data management activities

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

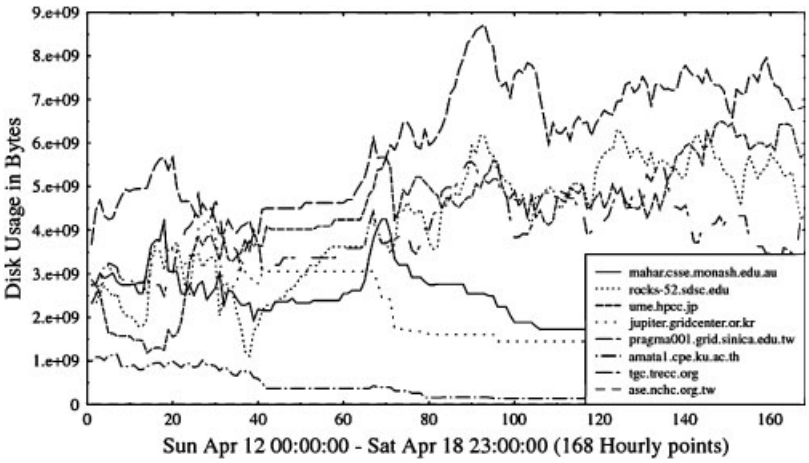


Figure 7. Disk space usage patterns.

as disk usage grows and shrinks. An increase in the y-coordinate represents data accumulation, whereas a decrease represents file removal. In the Savannah experiment, the disk space usage was directly proportional to the number of jobs executing on the resource. Therefore, a low y-coordinate (disk usage) value in the graph means that fewer jobs were executed on that resource at that moment. An oscillation shows that the data are being generated and removed uniformly. The job vs data relationship is important in order to understand the following behavioural analysis.

It can be observed that before the 70th hour (approximately), all resources had a smooth oscillatory behaviour, but the disk usage of jupiter.gridcenter.or.kr, mahar.csse.monash.edu.au and amata.1.cpe.ku.ac.th decreased afterwards, because these resources became unavailable. The constant disk usage at the end of the graph is due to the presence of common files that were retained on the resources for subsequent use by other computations. Owing to resource unavailability, the jobs were distributed among other available resources, which is evident from the increase in their disk usage patterns. Note how resources such as ume.hpcc.jp have shown a more noticeable increase in disk usage compared with others. This happened because these resources had a higher number of computational nodes available, which made them a preferable choice for job allocation. Note as well that the oscillation in disk usage curves became quite uniform after the 100th hour when the load was distributed again properly. This also illustrates the effectiveness of the garbage collection technique.

4.3. Trust management

As mentioned before, we had 90 parametric instances of the main model, which were broken into a sequence of 252 sub-computations and interconnected by the state files (see Figure 3). In this set-up, every successive sub-computation required the state file generated by its immediate predecessor as its input for execution. Each line in Figure 8 represents the first 20 sub-computations of randomly chosen parametric instances. It can be observed that sub-computations, belonging to a particular

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

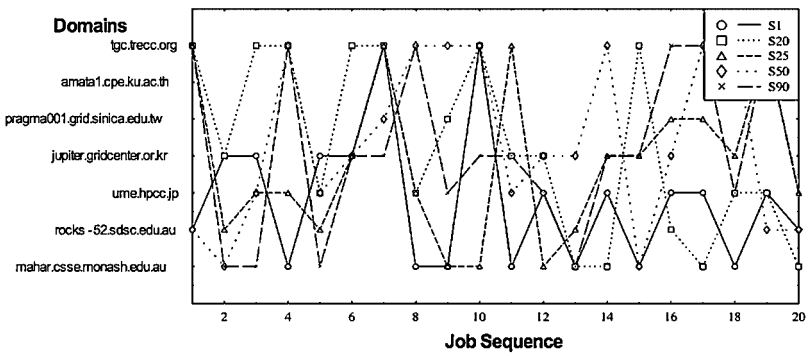


Figure 8. Inter-domain third party file transfers.

instance, were scheduled across seven different resources with exclusive CA domains. The situation required the state files to be copied accordingly (more resources were added at a later stage of the experiment). For example, to execute the 12th sequence of the 25th parametric instance on mahar.csse.monash.edu.au, the state file generated by the 11th sequence was copied from domain A (tgc.trecc.org) to domain B (mahar.csse.monash.edu.au). We performed file transfers using our third party communications mechanism, which allowed us to maximize the usage of the multi-VO Grid testbed.

The random load-balancing scheduler was used in the experiments. This was because certain mechanisms required by more complex schedulers to gather network and load statistics were not uniformly available across the testbed. For example, NWS was not supported on PRAGMA resources at that time, resulting in the under- or non-allocation of those resources by the scheduler. Other means of obtaining statistical data were also thwarted in various ways, given the non-uniformity of the resources available to us on the multi-VO testbed. Such issues are not unexpected in such a large and heterogeneous collection. It was therefore impossible for us to rely on the more sophisticated algorithms such as XSufferage. The latter would have been ideal as this algorithm considers both estimated file transfer time from source to target (resource) and the estimated execution time. This is crucial as rescheduling a job means migrating both the job and its input files. However, our attempts to use XSufferage failed to deliver optimal schedules given the lack of accurate statistical information, despite attempts to modify it to suit the suboptimal situation. We therefore employed our scheduling technique that was mostly but not completely arbitrary in allocating resources. Jobs generated from task-splitting were actually sorted by priority first, and then they were scheduled randomly on resources other than where their respective parents executed. This is not ideal but nevertheless achieves a satisfactory distribution across the whole testbed.

In any case, had XSufferage or similar APST algorithms been used, the fact is that third party transfers would still be required, simply because the resources on this testbed come from independent CA domains. The only reason for such transfers to become unnecessary is if all the tasks were scheduled within one VO, something that would reflect a poor degree of distribution and load balancing. Using the random load balancer, we achieved an acceptable level of resource utilization across all VOs involved.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



5. SOME LESSONS LEARNED

Apart from the strategies and solutions discussed above, other problems were identified in the course of the climate study described in this paper. Some of the lessons we have learned are briefly presented below:

- *Task specification*: Application specification languages should be developed, which could succinctly describe complex e-science problems and simplify the specification of PSAs. For PSAs, the specification of parametric instances should be intuitive, as in our planfile syntax, rather than requiring an independent specification for each parametric instance, in the same way that one might specify independent and unrelated jobs. Although the utilization of parameters and seqameters as indices of input and state files may seem unconventional, it does present a very concise mechanism that lends to easy use for e-science applications.
- *Application deployment*: The complexity of Grid testbeds should be hidden, such that the deployment of PSAs should not require significant changes to the way the applications are specified or launched. We sought to achieve this with the very simple design of the planfile syntax, even when we included seqameters: there is no marked difference in deployment despite the ability to perform task-splitting across sequential subtasks.
- *Garbage collection*: Garbage collection and relocation are equally important under storage constraints. It is very common in the e-science domain to perform garbage collection manually. However, it should be an integral part of the middleware itself, as it is aware of the data dependencies among the tasks precisely.
- *Trust delegation*: The trust delegation layer implemented in what we consider lower middleware, Globus, should be extended to provide support for third party communication. The solution provided in this paper can be considered a high-level solution, given that Nimrod/G, one layer above Globus, provides the third party transfer mechanism. Similar methods would have to be integrated with similar middleware. However, if Globus and other lower middleware could support such a mechanism, then upper middleware layers can focus on other issues, such as scheduling.

The above lessons clearly point towards more work in this problem space, constituting practical work as well as fundamental research for the future. Based on the above lessons, we have set new goals as described in Section 7.

6. RELATED WORK

A number of projects complement our work. The Apples Parameter Sweep Template (APST) [17] is a parametric modelling tool that allows the scheduling of complex workflows on the Grid, specified using large APST XML descriptions [18]. Such descriptions are unlike the simple planfile syntax developed for Nimrod/G and subsequently used by GridBus [7]. The APST scheduling model also assumes unlimited disk storage and single VO testbeds. GridBus' core technology is quite similar to that of Nimrod/G, but the former assumes unlimited disk storage and single-domain testbeds. ICENI [16] provides a dynamic service-oriented framework for Grid environments that are restricted to single domains, without any task-splitting and garbage collection.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a new task, data and trust management techniques to deploy large-scale applications on the multi-VO Grid testbed. These techniques were implemented during the first phase of the experiment involving the execution of large PSAs to study how the burning of Savannah affects the climate of Northern Australia. The implemented techniques include

- A task-splitting and check-pointing approach to break up large PSAs into a sequence of dependent subtasks for better fault recovery and improved application performance.
- A garbage collection technique to delete unnecessary data to comply with storage limits.
- A third-party file communication technique that delegates trust to agents deployed across multiple VOs to perform cross-domain file transfers.

Results show that Nimrod/G migrates sequential sub-computations to better resources when failures occur, while still executing as many tasks in parallel as possible. Moreover, disk usage patterns on computational resources illustrate that Nimrod/G efficiently co-schedules multiple computations on a single resource. However, Nimrod/G restricts the amount of cumulative data generated by these computations within permitted storage limits. Further results show that while migrating the downstream sub-computations across resources of multiple VOs, Nimrod/G effectively copies the state files along with other input files required by individual sub-computations to appropriate resources.

Future work we will set out to undertake include the following:

- specialized CPU and network state monitoring to optimize task scheduling;
- an intelligent back-up and restore mechanism to minimize state back-up costs and to improve fault recovery;
- implementation of novel clustering-based heuristics to optimize scheduling of dependent tasks and
- development of new models of computation for describing scientific applications, and new scheduling algorithms for exploiting concurrency between application components.

ACKNOWLEDGEMENTS

The Australian Research Council under an e-Research special initiative grant supports this work. We wish to thank Amanda Lynch and her colleagues for their assistance in running the climate study. We also acknowledge the CSIRO Division of Atmospheric Research who provided the computational models. We thank our PRAGMA partners for the resources used in the testbed.

REFERENCES

1. Abramson D, Giddy J, Foster I, Kotler L. High performance parametric modeling with Nimrod/G: Killer application for the global Grid? *International Parallel and Distributed Processing Symposium* (*IPDPS*), Cancun, Mexico, May 2000; 520–528.
2. Amsden A, Ramshaw J, Rourke OP, Kiva DJ. A computer program for 2- and 3-dimensional fluid flows with chemical reactions and fuel sprays. *LASL Report*: *LA-10245-MS*, Los Alamos National Laboratory, 1985.
3. Basney J, Livny M, Mazzanti P. Harnessing the capacity of computational grids for high energy physics. *Conference on Computing in High Energy and Nuclear Physics*, Padova, Italy, 2000.

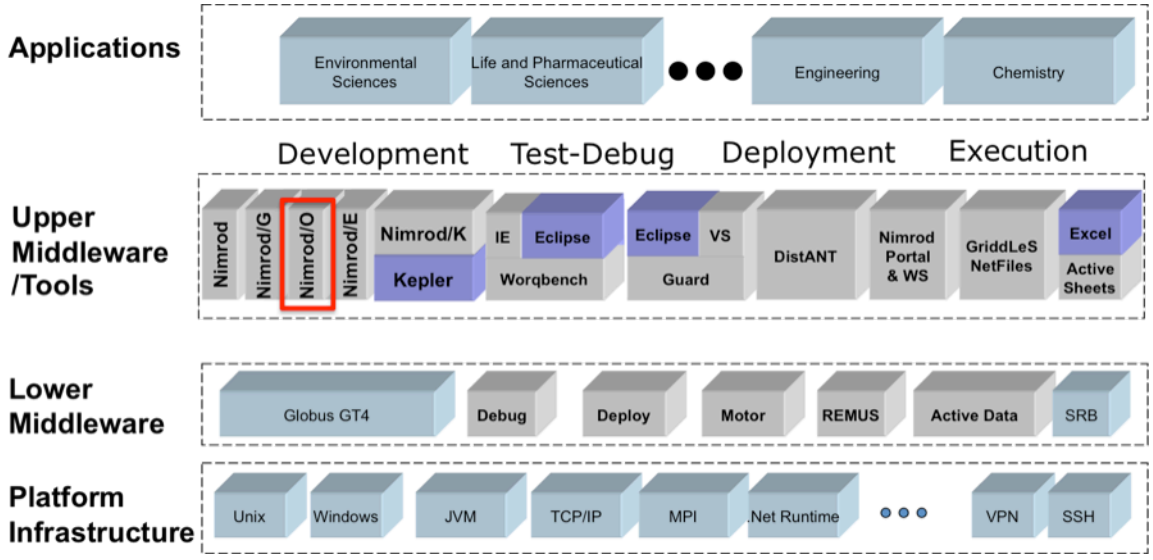
Copyright © 2008 John Wiley & Sons, Ltd.



4. Rogers S. A comparison of implicit schemes for the incompressible Navier–Stokes equations with artificial compressibility. *AIAA Journal* 1995; **33**(10):2066–2072.
5. Stiles JR, Bartol TM, Salpeter EE, Salpeter MM. Monte Carlo simulation of neuromuscular transmitter release using MCell, a general simulator of cellular physiological processes. *Computational Neuroscience*, Bower JM (ed.). Plenum Press: New York, 1998; 279–284.
6. Casanova H, Berman F, Zagorodnov D, Legrand A. Heuristics for scheduling parameter sweep applications in Grid environments. *Ninth Heterogeneous Computing Workshop* (*HCW00*), Cancun, Mexico, May 2000; 349–363.
7. Buyya R, Venugopal S. The Gridbus toolkit for service oriented grid and utility computing: An overview and status report. *The 1st IEEE International Workshop on Grid Economics and Business Models* (*GECON 2004*), New Jersey, 2004.
8. Tamir Y, Sequin CH. Error recovery in multicomputers using global checkpoints. *Proceedings of the International Conference on Parallel Processing*, Bellaire, MI, 1984; 32–41.
9. Meyer LAVC, Annis J, Wilde M, Mattoso M, Foster I. Planning spatial workflows to optimize grid performance. *ACM Symposium on Applied Computing*, *Proceedings of the 2006 ACM Symposium on Applied Computing* (*SAC 2006*), Dijon, France, 2006; 786–790.
10. Zheng C, Abramson D, Arzberger P, Ayyub S, Enticott C, Garic S, Katz M, Kwak J, Lee BS, Papadopoulos P, Phatanapherom S, Sriprayoonsakul S, Tanaka Y, Tanimura Y, Tatebe O, Uthayopas P. The PRAGMA testbed-building a multi-application international Grid. *Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops* (*CCGRIDW'06*), Singapore, May 2006; 57.
11. McGregor JL, Nguyen KC, Katzfey JJ. Regional climate simulations using a stretched-grid global model. *Report No. 32*, *WMO/TD-No. 1105*, Research Activities in Atmospheric and Oceanic Modelling, Geneva, 2002; 3.15–3.16.
12. Beringer J, Hutley L, Lynch A, Siems S, Tapper N. The Savanna Fire Experiment (SAFE). http://arts.monash.edu.au/ges/research/climate/fire/safe.php [1 May 2008].
13. Buyya R, Abramson D, Giddy J. An evaluation of economy-based resource trading and scheduling on computational power Grids for parameter sweep applications. *Workshop on Active Middleware Services* (*AMS 2000*) (in conjuction with *Ninth IEEE International Symposium on High Performance Distributed Computing*). Kluwer Academic Press: Dordrecht, 2000.
14. Abramson D, Buyya R, Giddy J. A computational economy for Grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems* 2002; **18**(8):1061–1074.
15. Tan J, Abramson D, Enticott C. Bridging organizational network boundaries on the grid. *IEEE Grid 2005*, Seattle, November 2005.
16. Gulamali MY, McGough AS, Newhouse SJ, Darlington J. Using ICENI to run parameter sweep applications across multiple Grid resources. *Case Studies on Grid Applications Workshop*, *Global Grid Forum 10*, Berlin, Germany, March 2004.
17. Casanova H, Obertelli G, Berman F, Wolski R. The AppLeS parameter sweep template: User-level middleware for the Grid. *Proceedings of the Super Computing Conference* (*SC'2000*), Dallas, TX, 2001.
18. Casanova H. APST XML Man Page, 2001, http://grail.sdsc.edu/projects/apst/apst_xml_man.html.
19. Buyya R, Abramson D, Murshed M. A deadline and budget constrained cost-time optimization algorithm for scheduling task farming applications on global Grids. *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, 2002.
20. Wolski R, Spring NT, Hayes J. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems* 1999; **15**:757–768.
21. Massie ML, Chun BN, Culler D. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing* 2004; **30**(7):817–840.
22. Chaitanya B, Moore R, Rajasekar A, Wan M. The SDSC storage resource broker. *Proceedings of CASCON'98*, 30 November–3 December, Toronto, Canada, 1998.
23. The Portable Batch System (OpenPBS). http://www.pbsgridworks.com/PBSTemp1.3.aspx?top_nav_name=Products&item_name=OpenPBS&top_nav_str=1 [1 May 2008].
24. Gentzsh W. Sun grid engine: Towards creating a compute power grid. *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2001; 35.
25. Hendriks EA. BProc: The Beowulf distributed process space. *The 16th International Conference on Supercomputing*, New York, NY, 2002; 129–136.
26. O'Callaghan DW, Coghlan BA. Bridging secure WebCom and European DataGrid security for multiple VOs over multiple Grids. *Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks* (*ISPDC/HeteroPar'04*), Washington, DC, 2004; 225–231.

Copyright © 2008 John Wiley & Sons, Ltd.

Chapter 4: Nimrod/O and optimization



The parametric computing model discussed in Chapters 2 and 3 requires complete state space enumeration, which is infeasible when there are many parameters. In this chapter we discuss the use of non-linear optimization algorithms to guide the search. A tool called Nimrod/O demonstrates these ideas, and is reported in papers (12) to (19).

The first use of these techniques in paper (12) shows how we constructed an air quality decision support system. In (13) through (16) we show how the ideas can be used in other engineering problems, and discuss the general architecture of Nimrod/O. Papers (17) – (19) discuss three search algorithms and their implementation in Nimrod/O.

(1) Lewis, A, Abramson, D and Simpson, R., "Parallel non-linear optimization : Towards the design of a decision support system for air quality management", IEEE Supercomputing 97, San Jose, 1997. 4-1 .. 4-12

(2) Abramson D, Lewis A, Peachey T, Fletcher, C., "An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics", SuperComputing 2001, Denver, Nov 2001. 4-13 .. 4.24

(3) Abramson, D, Lewis, A. and Peachy, T., "Nimrod/O: A Tool for Automatic Design Optimization", The 4th International Conference on Algorithms & Architectures for Parallel Processing (ICA3PP 2000), Hong Kong, 11 - 13 December 2000. 4-25 .. 4-36

(4) Peachey, T., Abramson, D., Lewis, A., Kurniawan, D. and Jones, R. "Optimization using Nimrod/O and its Application to Robust Mechanical Design", PPAM 2003, Fifth International Conference on Parallel Processing and Applied Mathematics, Czestochowa, Poland, Lecture Notes in Computer Science, Volume 3019 / 2004, pp. 730 – 737, ISBN: 3-540-21946-3, September 7-10, 2003. 4-37 .. 4-44

(5) Abramson, D, Lewis, A. and Peachy, T. "Case Studies in Automatic Design Optimisation using the P-BFGS Algorithm", 2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference, April 22-26, 2001, pp 104 – 109, Seattle, Washington (USA). 4-45 .. 4-50

(6) Peachey, T. C., Abramson D., and Lewis A. "Parallel line search", in Optimization: Structure and Applications, Springer Optimization and Its Applications , Vol. 32, Pearce, Charles; Hunt, Emma (Eds.), 2009, ISBN: 978-0-387-98095-9. Chapter 20, pp 369 – 381. 4-51 .. 4-63

(7) Lewis, A. Abramson, D. and Peachey, T., "RSCS: A Parallel Simplex Algorithm for the Nimrod/O Optimization Toolset", Scientific Programming, 14(1), 1-11 (2006). IOS Press. 4-64 .. 4-74

(8) Lewis, A. and Abramson, D. "An Evolutionary Programming Algorithm for Multi-Objective Optimisation", Proc. 2003 Congress on Evolutionary Computation (CEC 2003), vol. 3, pp. 1926-1932, Canberra, Dec. 2003. 4-75 .. 4-81

Contributions

Lewis was a PhD student under my supervision; Peachey was employed as a research associate under my supervision; Fletcher contributed a case study in aerodynamics; Simpson assisted with a case study in air pollution management.

Parallel Non-Linear Optimization : Towards The Design Of A Decision Support System For Air Quality Management

Andrew Lewis
Queensland Parallel Supercomputing Facility
Division of Information Services
Griffith University
Brisbane, QLD, 4111
Australia
A.Lewis@its.gu.edu.au

David Abramson
Department of Digital Systems
Monash University
Clayton, VIC, 3168
Australia
davida@dgs.monash.edu.au

Rod Simpson
Faculty of Environmental Sciences
Griffith University
Brisbane, QLD, 4111
Australia
R.Simpson@ens.gu.edu.au

Abstract

Large numerical simulation codes have been applied to a wide range of scientific and engineering problems. In the environmental arena the ability to predict the results of certain scenarios by computational science has allowed the choice of strategies which maximize desired outcomes (e.g. financial return) whilst minimizing environmental damage. Access to high performance computing resources has focussed attention on the development of environmental decision support systems which can be used by regulatory agencies and industry planners in evaluating different policy options. A common objective is to find a solution which optimizes some pre-defined criteria. In environmental modelling, the type of optimization problems which need to be considered involve non-linear cost functions over both discrete and continuous parameter values.

In this paper we address the optimization component of a decision support system, and perform some initial benchmark studies to assess the effectiveness of the overall

Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

(c) 1997 ACM 0-89791-985-8/97/0011 $3.50

approach**.** The algorithm selected for initial study is based on the quasi-Newton BFGS method. Whilst the BFGS algorithm is generally implemented sequentially, because of the focus of the decision support systems described in the paper we are interested in parallelizing the basic algorithm. This is achieved by concurrent evaluation of functions in finite difference approximations to the derivative and a method of interval subdivision in simple bound constrained line searching.

In a realistic problem of air quality management, use of the parallel optimization algorithm as part of an optimizing decision support system is shown to have significant performance gains over other methods of solution. In initial tests it uses less than half the evaluations of a computationally demanding numerical simulation previously used simple enumeration techniques require and is four times faster than traditional sequential optimization methods. This case study has successfully demonstrated the application of an optimization system to a core environmental model, and the feasibility of its use to solve real world problems using parallel and distributed supercomputers.

Keywords

non-linear optimization, parallel optimization, numerical simulation, decision support systems, environmental management

1. Introduction.

Large numerical simulation codes have been applied to a wide range of scientific and engineering problems [1]. In the environmental arena the ability to predict the results of certain scenarios by computational science has allowed the choice of strategies which maximize desired outcomes (e.g. financial return) whilst minimizing environmental damage. In the area of air pollution, such technology has been used for real world studies predicting air pollution levels over a planning horizon, and then examining control strategies which are realistic [2,3,4].

Access to high performance computing resources has focussed attention on the development of environmental decision support systems which can be used by regulatory agencies and industry planners in evaluating different policy options [5]. A common objective is to find a solution which optimizes some pre-defined criteria. The problems may be posed as questions such as "What set of control parameters will minimize production of smog over the planning horizon?" or "How can we minimize the financial burden of maintaining a desired air quality standard?" These are the straightforward uses for which optimizing decision support systems are intended.

Performing real world optimization using computational models is computationally very demanding. Parallel and distributed supercomputing is generally seen as the only cost effective pathway for the delivery of timely results. It is routinely used in the preparation of meteorological forecasts, an application which whilst under considerable pressure to deliver results in short time frames, is possibly orders of magnitude less demanding in terms of computing cycles required. An optimal solution to an environmental problem may not need to be delivered with less than 24 hour turnaround, but it should be available before the reasons for the question are forgotten! It is for this reason that a major focus of the work described in this paper

is parallelization of the optimization algorithms used for solving these problems.

There are also a number of interesting optimization problems during the development of such an environmental decision support system**.** Modelling photochemical pollution has a number of essential components. Meso-scale meteorological flows can have a profound effect on smog production. Successful simulation of photochemical smog production in turn requires the accurate simulation of these air flows. Over the years, a number of prognostic mesoscale wind field models have been developed to compute 3-dimensional wind fields [6,7]. Wind field models are based on the dynamics and thermodynamics of large scale flow, as described by the Navier Stokes equations. Physical processes, such as the effect of solar and longwave radiation and interactions with the surface, must also be included. It is these effects in particular that may require considerable tuning to minimize the variance between computed and measured wind fields. This provides another opportunity for application of optimizing systems to internally verify the environmental model itself.

A decision support system can thus be applied in a number of different ways and, if engineered properly, will serve all these purposes using one architecture.

2. The Architecture of an Optimizing Decision Support System.

The architecture of the envisaged decision support system is shown in Figure 1. It will have the following characteristics:

- It should utilize high performance parallel and distributed platforms, as the most cost effective way of providing sufficient computing cycles to perform real world optimization using computational models.

- It should incorporate a wide range of optimization algorithms where the cost of a solution is computed by a modelling application, to tackle the various different types of optimization problems encountered in environmental modelling.

- It should allow for rapid prototyping of an experiment, since users of such systems often like to experiment with different aspects of the model and explore a wide range of scenarios.

- It should incorporate user interaction in the decision making process. Since it is often difficult to formulate a good cost measure for some environmental problems, it is useful if the user can guide the system using domain specific knowledge.

In this paper we only address the optimization component of the architecture, and perform some initial benchmark studies to assess the effectiveness of the overall approach**.** While numerical non-linear optimization is a large, well established field [8,9,10,11], combining optimization with computational models is quite recent. In general, **e**nvironmental models are computationally expensive, and thus high performance computing platforms are required in order to deliver solutions in a timely manner. The development of parallel algorithms for the tasks to be completed is thus an integral part of system development.

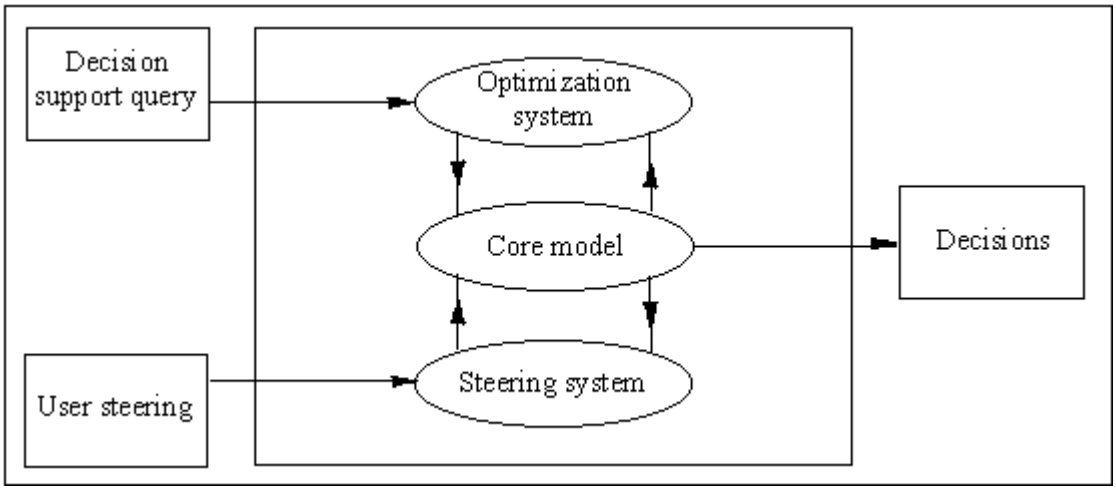


Figure 1. Decision Support System Architecture

In environmental modelling, the type of optimization problems which need to be considered involve non-linear cost functions over both discrete and continuous parameter values. Many robust codes for such purposes have been developed. Finding optimal solutions to discrete problems in which the cost is computed by a numerical simulation is extremely demanding. Meta-heuristics like simulated annealing [12], genetic algorithms [13], and Tabu search [14], are candidates for searching the parameter space. However, these can be computationally expensive in their own right and the difficulties involved in using them with expensive application models and integrating them with continuous function solution requires further investigation.

For continuous functions, gradient descent methods use the derivative of the cost function, as well as its value, to select a search direction, essentially reducing the multivariate optimization problem to univariate minimization along a search vector; a line search. They thus consist of two main operations which are executed repeatedly, gradient calculations and line searching. The algorithm selected for initial study is based on the quasi-Newton BFGS method, widely regarded as one of the most efficient and robust gradient descent methods for use with continuous functions[8,9,15].

When the cost function is the computed result of a numerical simulation, as proposed in this paper, finite difference approximations to the derivative must usually be employed. Whilst the BFGS algorithm is generally implemented sequentially, because of the focus of the decision support systems described above we are interested in parallelizing the basic algorithm. The gradient information gathering phase of the algorithm appeared a promising candidate for parallelization. The degree of parallelism achievable is limited, however, by the dimensionality of the problem domain.

For unconstrained optimization the line search phase is often inherently sequential, but for the class of problems under consideration there will often be restrictions on the expected range of variables determined by physical constraints on their possible values. The restricted expected range of the variables allows the formulation of these problems as nonlinear optimization

problems with simple bounds, and so the line search phase of the algorithm may be parallelized by using a method of interval subdivision, rather than the typically used sequential stepping methods. A brief outline of the parallel algorithm is given in Appendix A. A detailed description is given elsewhere [16].

3. Air Quality Management – An Initial Benchmark

In the target domain of air quality management, one of the major uses of photochemical airshed models is to compute oxidant concentrations. Oxidants, such as ozone, are generated as a result of the chemical interaction between various precursors such as oxides of nitrogen (NOx) and other reactive organic compounds (ROCs) in the presence of ultra-violet radiation. Ozone is of particular importance because of its health related side effects; ozone levels in Australian urban areas recently have been observed to exceed 0.12 ppm, which is a widely adopted health standard level. Results in this area are of considerable and immediate interest to our environmental research colleagues.

The prototype parallel optimization code is being used with the CIT model developed by McRae et al [17], to compute the transport and production of photochemical smog within an urban airshed. As such it forms one half of the decision support architecture described in this paper. The addition of user steering is to be part of future development work, and may draw on other research to support this function [18]. The system is currently being used to determine the sensitivity of oxidant photo-chemistry to various input parameters.

The benchmark described in this paper was based on detailed meteorological and pollution emission data for a particular scenario have been collected for the Australian city of Perth. Further sampling studies are being performed for a number of other Australian cities. The model behaviour is well understood, as it has served as the basis for earlier work including simple enumeration studies on parallel and distributed computing platforms [19]. As an initial benchmark for the work the optimization algorithm is used to determine the NOx and ROC concentrations, as input parameters, that would produce a minimal peak hourly average ozone concentration over a 24 hour period. Future studies are expected to require minimization over up to 10 different input parameters, and objective functions related to overall population exposure to a variety of chemical species.

Figure 2 shows a sample ozone contour for the model region. It clearly shows the non-linear effect of varying ROCs and NOx on the ozone concentration. In some regions of the control space, increasing one of the precursors can increase the ozone, and in others it can decrease the ozone. Thus, a simplistic strategy of decreasing the precursors may not have the desired effect of decreasing the oxidants. It is this effect that makes it essential to correctly model the process when evaluating control strategies.

The parameter ranges are limited to a subset of the whole domain to exclude the trivial solution of reducing all precursor concentrations to zero. These limits were initially arbitrarily set to 50% in each dimension, but the range permissible for NOx was expanded down to 25% to allow the distinct non-linear feature seen in Figure 2 to extend across the whole domain. The desired solution accuracy is arbitrarily set to ±5% in the input parameters, a reasonable estimate for the

accuracy of the simulation.

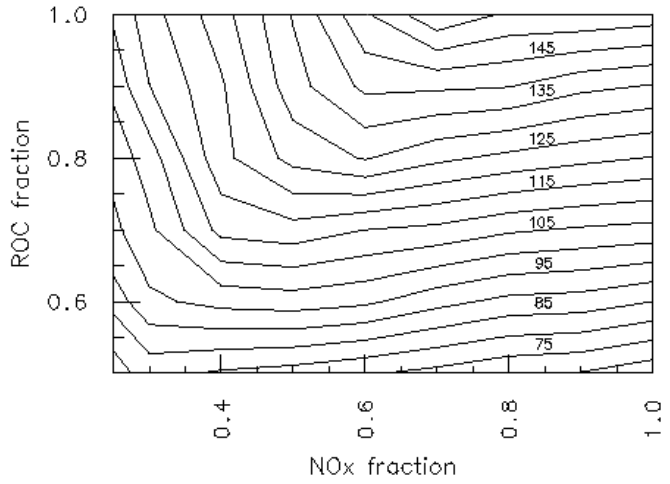


Figure 2. Typical surface: Ozone concentration (ppb) as a function of NOx and ROC

The initial benchmark was run on an IBM SP2 supercomputer. Since the problem was only 2-dimensional, the gradient determination used a maximum of 4 CPUs. This is because central difference approximations are used for gradient calculation, performing two function evaluations for each of two dimensions, giving a total of four independent operations. The configuration of the supercomputer places a current practical maximum of 10 CPUs on the line search phase of optimization. Since the machine is a shared facility, this maximum number of CPUs may not be allocated to every line search request.

During the period of this trial an average 4.6 CPUs were utilized. The parallel distribution of function evaluations was prototyped in Linda and the separate gradient calculation and line search tasks were submitted via system calls to the IBM LoadLeveler batch queueing system with appropriate requests for parallel computing resources for each task.

A solution set of input concentrations of NOx and ROC that minimized generated ozone was found at 100% NOx and 50% ROC. Inspection of Figure 2 confirms this to be the global minimum for the search domain. This result was achieved in under 14 hours of wall-clock compute time (ignoring queue wait time). This represents 46 evaluations of the cost function, using a simplified chemistry [20] in the airshed model to reduce the computational load in these exploratory tests. Each run of the airshed model currently takes about 1 hour 20 minutes of CPU time. A comparable solution by simple enumeration would require an estimated 140 hours of computing.

The same problem was solved using an inherently sequential optimization algorithm. This algorithm uses a combination of bracketing the minimum by parabolic extrapolation followed by isolation of the minimum using Brent's method[21]. The parallel algorithm, as mentioned at the end of section 2, was modified to use an iterative interval subdivision method for line searching. Using a single CPU of the IBM SP the sequential algorithm required 51 hours of wall-clock

compute time, performing 39 evaluations of the cost function. The prototype parallel code thus achieved a speedup of 3.8 relative to the sequnetial algorithm. Using 4.6 CPUs this represents a parallel efficiency of 83%. The effect on speedup of the number of processors allocated to a line search, and thus the number of interval subdivisions performed concurrently, will be investigated in future work.

Unlike the parallel algorithm, the sequential algorithm terminated in a local minimum (79% NOx, 50% ROC) which was close to the global minimum actually discovered by the parallel algorithm. The method of line search adopted in the parallel algorithm has been found to more reliably find global minima in a number of test cases.

4. Discussion

In the example just discussed only 2 parameters are used in the optimization. As previously noted up to 10 parameters may need to be optimized in future studies. While the models in practice cannot be expected to be more than ±5% accurate, even at this accuracy for increasing n simple enumeration requires 10^n cost function evaluations. This rapidly becomes an infeasible method of solving problems of this type. The behaviour of the parallel optimization algorithm for increasing n is a subject of future work. The evidence of these early trials in which it uses far fewer function evaluations suggests it to be considerably more efficient than enumeration. Further tests are also required to compare its convergence rate with that of sequential optimization methods.

As indicated, the benchmark problem used an average 4.6 CPUs during the trial. There are practical limits to the number of CPUs this method can profitably utilize. During the gradient determination phase, two CPUs are required for each dimension, to perform the finite difference approximation. So any more than $2n$ processors are wasted for gradient determination. For the benchmark problem $n = 2$, so the maximum number of processors useful for gradient determination is 4.

During the line minimization phase, interval sub-division is repeated until the sub-intervals are less than the desired tolerance. So for a given fractional tolerance, t, the maximum number of CPUs that can usefully be employed is $1/t$. Since most line searches will be less than the entire span of the search domain, to a first approximation the average number of CPUs usefully employed will be $1/(2t)$. For the benchmark problem $t = 0.1$ so the number of processors useful for line minimization is 5.

For the benchmark problem, the number of processors available actually coincided with the theoretical optimum most of the time, as reflected in the high parallel efficiency. With some code refinements the parallel efficiency could be made very high because of the negligible extent of serial code, most computation being performed in parallel in the objective function evaluation. The algorithm scaling, however, is strictly limited by problem parameters.

Unlike the sequential algorithm, the parallel algorithm is not trapped in some local minima. This is illustrated in Figure 3 which shows the sequential algorithm using a combination of stepping and parabolic interpolation, and the parallel algorithm using interval sub-division. The sequential

algorithm terminates in the first minimum found, while the parallel algorithm, extending its search to the boundary, finds the lower minimum.

To date only simple bounds on the optimization domain have been considered. The end uses of the decision support system may involve more general constraints. Of particular note is the challenging extension to the case study described in this paper, where the objective becomes the minimization of the financial cost of achieving some desired standard of air quality. In this case the problem may be expressed as the minimization of a (possibly discontinuous) multivariate linear function, the financial cost of reductions in emissions, subject to non-linear constraints derived from numerically intensive simulations of smog generation.

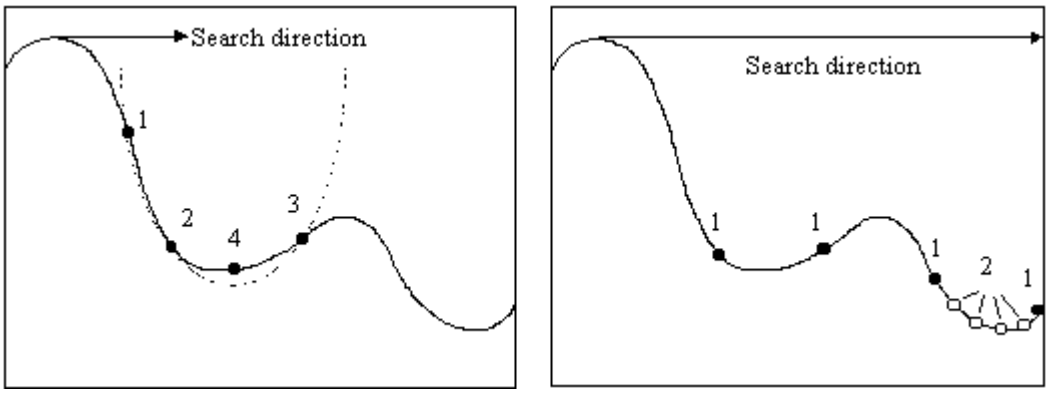


Figure 3: Comparison of sequential and parallel line minimization

The system architecture described in this paper is quite similar to that of the DAKOTA project at Sandia Laboratories [23] in the U.S.A. It additionally allows for interactive steering of the application which is important for the target application domains. DAKOTA has mainly been applied to problems in structural mechanics. The focus of the work reported here is on environmental applications so in this regard too, it is complementary to the work at Sandia Laboratories.

As the DAKOTA project has already shown with successful applications in container shape optimization, coating flow die design and vibration isolation system design, the techniques described in this paper have potential for much wider application than the initial environmental problems for which they are being developed. Interest has already been expressed in their use in vehicle aerodynamic drag minimization studies and prosthetics design in medical engineering.

5. Concluding Remarks.

Other authors [22] have demonstrated domination of computing time by user-supplied function-gradient evaluations. In the class of problems under study in this work, this trend reaches an extreme where the computational cost of the optimization algorithm is essentially negligible. In this light, the concurrent performance of function evaluations becomes of paramount importance.

In a realistic problem of air quality management, use of a parallel optimization algorithm as part of an optimizing decision support system has shown significant performance gains over other methods of solution. In initial tests it uses less than half the number of evaluations of a computationally demanding numerical simulation than previously used simple enumeration techniques require and is more than four times faster than traditional sequential optimization methods. This case study has successfully demonstrated the application of an optimization system to a core environmental model, and the feasibility of its use to solve real world problems using parallel and distributed supercomputers.

Future work is anticipated to improve the robustness of the methods in application on a wider range of problems. Applications other than those studied to date are anticipated to include discrete parameters as well as continuous variables, and the probable inclusion of a “floor function” termination criterion to facilitate go/no-go design decisions. Other issues involved in building a real decision support system, such as the user steering system, integration of visualization sub-systems and user interface will be part of future development work.

Acknowledgements.

The authors would like to thank their colleague Rok Sosic for his helpful suggestions tendered in a number of discussions and Martin Cope and Steven Spencer for their advice regarding airshed chemistry modeling. This work was performed as part of a collaborative project between Griffith University, Monash University, Queensland Department of Environment and Victorian Environment Protection Agency, funded in part by an ARC Collaborative Grant. Thanks go to the Queensland Parallel Supercomputing Facility for the use of their IBM SP.

References.

[1] G. Bell, **The Future of High Performance Computers in Science and Engineering**, Communications of the ACM, **32**(9), pp 1091-1101, 1989.

[2} M. Cope, F. Carnovale, and B. Cook, **An Air quality Modelling System of Oxidant Transport and Production in Port Phillip Control Region**, Proceedings of the 10th International Conference of the Clean Air Society of Australia and New Zealand, 1990.

[3] R.A. Harley, A.G. Russell, G.J. McRae et al**. Continued Development of a Photochemical Model and Application to the Southern California Air Quality Study (SCAQS) Intensive Monitoring Periods**. Carnegie Mellon University, Pittsburgh, PA and California Institute of Technology, Pasadena, CA. Report to the Coordinating Research Council under project SCAQS-8, 1992.

[4] G.J. McRae and A.G. Russell, **Effectiveness of the Air Quality Management Plan to Improve Air Quality**, report to the South Coast Air Quality Management District, El Monte, CA, 1989.

[5] B. Breckling and F. Müller, **Current trends in ecological modelling and the 8th ISEM conference on the state-of-the-art**, Ecological Modelling, **75/76**, pp 667-675, 1994.

[6] M.E. Cope and G.D. Hess, **Meso-scale meteorological modelling for input into a 3-dimensional photochemical oxidant model**, Proceedings International Congress on Modelling and Simulation, Perth, Australia, 1993.

[7] W.L. Physic, **Review: Mesoscale modelling in complex terrain**, Earth-Sci Rev., 25, 199-235, 1988.

[8] P.E. Gill, W. Murray, & M.H Wright, **Practical Optimization**, Academic Press, London, 1981.

[9] R. Fletcher, **Practical Methods of Optimization**, 2nd ed., Wiley, New York, 1987.

[10] P.E. Gill & W. Murray, eds., **Numerical Methods for Constrained Optimization**, Academic Press, London.

[11] D.G. Luenberger, **Introduction to Linear and NonLinear Programming**, Addison-Wesley, Reading, MA, 1973.

[12] N. Collins, R. Eglese and B. Golden, **Simulated annealing: An annotated bibliography**, American Journal of Mathematical and Management Sciences, **8**, Nos 3 & 4, pp 209-307, 1988.

[13] D. Goldberg, **Genetic Algorithms: In Search, Optimization and Machine Learning**, Addison-Wesley, 1989.

[14] F. Glover, **Tabu Search - part 2**, ORSA Journal on Computing, **2**(1), pp 4-32, 1990.

[15] W.H. Press, **Numerical Recipes: the Art of Scientific Computing**, Cambridge University Press, 1986.

[16] A.Lewis and D. Abramson, **A Parallel Algorithm for Bound Constrained Optimization of Numerical Simulations**, in preparation.

[17] G.J. McRae, A.G. Russell and R.A. Harley, **CIT photochemical airshed model -users manuals**, Carnegie Mellon University, Pittsburgh, PA and California Institute of Technology, Pasadena, CA, 1992.

[18] S.G. Parker and C.R. Johnson**, SCIRun: A Scientific Programming Environment for Computational Steering**, Proceedings of the 1995 ACM/IEEE Supercomputing Conference, San Diego, U.S.A., 1995.

[19] D.Abramson, M. Cope, and R. McKenzie, **Modelling Photochemical Pollution using Parallel and Distributed Computing Platforms**, Proc. PARLE-94, pp 478-489, Athens Greece, July 1994.

[20] M. Azzi, G. Johnson, and M. Cope, **An introduction to the Generic Reaction Set photochemical mechanism**, Proceedings of the 11th International Conference of the Clean Air Society of Australia and New Zealand, Brisbane,Australia, 1992.

[21] R.P. Brent, **Algorithms for Minimization without Derivatives**, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[22] J.J. Moré, **On the Performance of Algorithms for Large-Scale Bound Constrained Problems**, Large-scale Numerical Optimization, Ch. 4, SIAM, Philadelphia, 1990.

[23] M.Eldred, D.Outka, C.Fulcher, and W. Bohnhoff. **Optimization of complex mechanics simulations with object-oriented software design**, Proc. 36th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conf., pp 2406-2415, New Orleans, LA, April 1995.

Appendix A - Parallel Algorithm

Initialization

- Evaluate the objective function at the starting point
- Calculate the gradient at the starting point by finite difference approximation (parallel function evaluation)
- Set the inverse Hessian to the unit matrix
- Set the initial line direction to the inverse of the gradient

Perform line minimization

- Truncate the line search vector for active constraints
- Determine the nearest boundary in the search direction and set maximum excursion accordingly
- Sub-divide the interval as desired (usually an integer multiple of processors available)
- Evaluate the objective function at the sub-intervals (parallel function evaluation)
- Select a bracket of three points containing the minimum value
    - Yes - Line minimization complete
    - No - Sub-divide the bracket and repeat the evaluation

Test for convergence

If (step change in function value and largest step change in position in any dimension are less than the desired tolerance)
Then {Calculate the gradient at the "minimum"
  If (gradient is less than an empirically determined constant)
  Then optimization is complete
  Else {if (the "minimum" is at a boundary normal to the gradient)
    Then optimization is complete
    Else { if (change in function value over previous two steps is less than desired tolerance)
      Then optimization is complete
      Else {Reset the inverse Hessian to the unit matrix, reset the line search direction to the inverse of the gradient and **repeat line minimization**}
      }
    }
  }

Perform BFGS update

- Calculate a new gradient at the line minimum
- Calculate the step change in the gradient
- Apply the BFGS update to the inverse Hessian
- Calculate the new line search direction and **repeat line minimization**

An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics

David Abramson †, Andrew Lewis §, Tom Peachey †, Clive Fletcher ‡

† Computer Science & Soft. Eng.
Monash University,
CLAYTON, VIC 3800,
Australia

§ HPC Facility
Griffith University,
Nathan, Qld, 4111,
Australia.

‡ CANCES,
University of New South Wales
Australian Technology Park
Eveleigh NSW 1430
Australia

Abstract

In this paper we describe the Nimrod/O design optimization tool, and its application in computational fluid dynamics. Nimrod/O facilitates the use of an arbitrary computational model to drive an automatic optimization process. This means that the user can parameterise an arbitrary problem, and then ask the tool to compute the parameter values that minimize or maximise a design objective function. The paper describes the Nimrod/O system, and then discusses a case study in the evaluation of an aerofoil problem. The problem involves computing the shape and angle of attack of the aerofoil that maximises the lift to drag ratio. The results show that our general approach is extremely flexible and delivers better results than a program that was developed specifically for the problem. Moreover, it only took us a few hours to set up the tool for the new problem and required no software development.

Introduction

Computational science and engineering techniques have allowed a major change in the way that products can be engineered. Rather than building real world prototypes and performing experiments, a user can build a computational model that simulates the physical processes. Using such a model, many design alternatives can be explored computationally in a fraction of the time required. The technique has been applied widely in the areas of aviation, automotive engineering, environmental assessment and electromagnetics.

In the past, we have produced tools that assist a user in performing a rigorous design experiment using an arbitrary computational model. The Nimrod [1] and EnFuzion [2] tools make it possible to describe a number of discrete design scenarios using a simple declarative programming language. The system then produces discrete scenarios, each a unique combination of parameter values from the cross-product of the parameter ranges. If integer or floating point parameters are specified, a step count is used to discretise the domain. The scheme is very powerful, and has been used in real world problems [3]. In order to speed the execution of the experiment, distributed computers are used seamlessly to explore multiple scenarios in parallel. Whilst Nimrod and EnFuzion are optimized for clusters of computers, a "Grid Aware" version of Nimrod, called Nimrod/G [18], utilises resources on a global computational grid [4].

The biggest disadvantage of Nimrod is that when very large search spaces are specified, or when high resolution in the parameter values is required, the number of scenarios may exceed the computational power available. Further, the user may not actually want to explore all design space, but may be satisfied with a "good" solution instead. This background motivated the development of Nimrod/O, a variant of the Nimrod system that performs a guided search of the design space, rather than exploring all combinations. Nimrod/O allows a user to phrase a question like: "What set of design parameters will minimise (or maxmise) the output of my model?". If the model computes metrics like cost, lifetime, etc, then it is possible to perform automatic optimal design.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC2001 November 2001, Denver (c) 2001 ACM 1-58113-293-X/01/0011 $5.00

Nimrod/O system is almost unique in its ability to solve aribtrary problems without requiring the user to develop optimization code. In general, there are very few systems that allow a user to embed a computational model within a larger problem solving environment, let alone perform automatic optimization. Some systems which have some of these properties are DAKOTA, NEOS, NetSolve and SciRun. The DAKOTA project at Sandia laboratories investigated the combination of optimization with computational models [5]. DAKOTA is a C++ based tool kit that allows a user to choose different optimization algorithms to guide the search for optimal parameter settings. DAKOTA has been successfully demonstrated on structural mechanics applications. However, DAKOTA did not support rapid prototyping and still requires the construction of new programs for each different type of experiment which is performed. NEOS [6] is a distributed, web-based optimization engine that allows a user to solve optimization problems using special remote optimization servers. However, NEOS relies on the objective function being specified in an algebraic form and does not support objective functions implemented by CS&E models. Likewise, NetSolve [7] provides a web based engine for solving linear algebra problems, but does not explicitly address optimization using CS&E objective functions. A number of researchers have investigated the use of Genetic Algorithms for solving complex optimisation problems of the type discussed here, but they do not support multiple search algorithms in the same tool [24]. SciRun [8], and its follow-on Uintah [9], are interactive tools that allow a user to build a CS&E model very rapidly using a graphical programming interface. However, they do not support optimisation. *In contrast, Nimrod/O combines optimization, distributed computing and rapid prototyping in one tool.* The only tool we have found that shares the idea of combining optimisation and rapid prototyping is Optimus, a commercial package produced by Numerical Technologoies. Some details of this are available on the company web site [25].

In this paper we will discuss the Nimrod/O system and its application to a real world case study, the design of a simple, but optimal, aerofoil. We begin with a general discussion of automatic design optimization and the challenges it poses, and then expose the Nimrod/O architecture. We compare the performance of Nimrod/O at solving the aerofoil design against a tailored solution built in Fortran.

Searching for Optimal Designs

Design optimization is not new. There are many examples, particularly from the operations research literature, of the use of optimization theory to find good solutions to real world problems [10]. However, almost all of this work has assumed that the objective function can be expressed algebraically. This means that it is possible to evaluate the function quickly when a new set of design parameters are generated. Further, it is often possible to differentiate the function, which assists algorithms that try to perform gradient descent.

We are concerned with designs that are so complex that their effectiveness can only be evaluated by running a computational model. An important design goal is that the exact nature of the model is not important, and may be solved by a discrete event simulation or the solution of a set of partial differential equations. Because of this generality, it is necessary to run the model from the beginning each time the parameters are changed. Further, if derivatives of the objective function are required, these are typically calculated using a finite difference approximation [11]. Because of this, the cost of executing the optimization algorithm is almost totally dominated by the cost of running the computational model.

From a user's perspective, the problem can be phrased simply - minimise (or maximise) the objective function across a set of parameters. Because almost all real world problems have bounds on the legal parameter values, it is possible to bound the search by these limits. Further, it is often necessary to place further *constraints* on the solution. For example, additional functions that combine parameter values can be used to further reduce the domain. We allow a user to specify both *hard* and *soft* constraints. Hard constraints are enforced during the search process itself. If a hard constraint would be violated by a particular choice of parameter values, then that part of space is not explored. In contrast, soft constraints are implemented by adding a penalty value to the objective function. Accordingly, soft constraints can be violated during the search, but the objective function is artificially higher than if the constraint was satisfied. These techniques are standard in non-linear optimization [12].

In general, the objective functions under consideration are non-linear, may not be smooth and may contain a high degree of noise. In addition, parameters may be continuous or discrete, depending on the nature of the underlying

problem. Thus, no single optimization procedure will work for all problems. Some problems will contain multiple local minima, and it is impossible to guarantee that any one search algorithm will find the global minimum. Accordingly, Nimrod/O supports a range of algorithms, which can be executed multiple times (in parallel) from different starting locations. When a number of algorithms are used, each with different starting locations, it is possible to gain some insight into the nature of the objective function, and to generate a number of potential solution to the problem. Regardless of the search technique that is used, we assume that the computational model is well formulated, stable and robust across the parameter ranges.

Nimrod/O Search Algorithms

At present, we have implemented four optimisation algorithms, namely a gradient search code called P-BFGS [13][12], a Simplex search [20], a Divide-and-Conquer heuristic [13] and Simulated Annealing [23]. In this section we give a brief introduction to each of these algorithms. More details can be found in [13][14]. In the case study reported in the paper we only used two of these algorithms, P-BFGS and Simplex.

P-BFGS

The BFGS optimization method is a quasi-Newton method, that is it is based on the iterative formula

$$x_{k+1} = x_k - \alpha_k H_k g_k . \qquad (1)$$

Here $x_0, x_1, \ldots$ are the successive approximations to the optimal point, considered here as column vectors. g_k is the gradient of the objective function at the point x_k (also a column vector) and H_k is a square matrix described below. The scalar α_k is a positive number chosen so as to minimize the objective $f(x_{k+1})$. The initial value of the matrix, H_0, is the identity matrix, so (1) becomes $x_1 = x_0 - \alpha_0 g_0$ which implies a search directly downhill from x_0. As the search proceeds H_k is updated so that it approaches the "inverse Hessian" matrix. Then (1) contains second derivative information, giving faster convergence than a downhill search, at least if the objective function is sufficiently smooth. The quasi-Newton methods vary as to the updating formula for H_k. For the BFGS method this is

$$H_{k+1} = H_k + \frac{1}{g_k^T c_k}[b_k c_k c_k^T - c_k g_k^T H_k - H_k g_k c_k^T] \quad (2)$$

where $b_k = 1 + (g_k^T H_k g_k)/(g_k^T c_k)$.

We have implemented a parallel variant in which the elements of the finite difference stencil are computed in parallel and the line search is broken into a number of parallel evaluations. The latter modification alters the properties of the algorithm from the original BFGS [12]. BFGS can be applied to both continuous and discrete domains, and works well when the surface is smooth. The concurrency in PBFGS is dependent on the number of parameters and the degree of subdivision of the line search. Typically, we have tested with about 10 processors and have achieved reasonable speedups. When multiple starts are considered, the concurrency is higher.

Simplex

The simplex algorithm implemented in Nimrod/O is a variant of the simplex method of Spendley, Hext and Himsworth [27][20].

The algorithm proceeds as follows. For a search in N dimensional space, the method maintains an N dimensional "simplex", that is a set of $N+1$ points which do not fit into any hyperplane. For example in 2 dimensions, a simplex is a set of 3 points not in a straight line, the vertices of a triangle. In 3 dimensions a simplex is a tetrahedron. Basically the method iteratively replaces the worst vertex by another on the other side of the vertex.

The cost is evaluated at the points of the simplex. Suppose we label the vertices in order of descending cost: x_0, x_1, ... x_m. For convenience we will write *w* for x_0 (the worst), *n* for x_1 (next worst), and *b* for x_m (the best). Let the point *c* be the centroid of all the vertices except *w*. We take *w* and reflect it through *c*, to create new points:

p	an equal distance on the other side of *c*
q	half the distance of *w* from *c*
r	twice as far from *c* as *p*.

These points are all candidates to replace *w* in the simplex, as shown below in Figure 1.

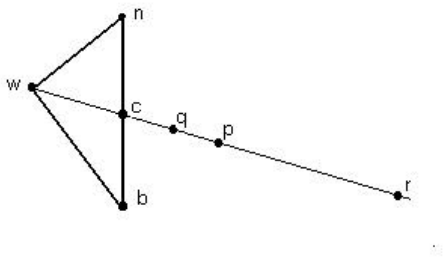


Figure 1 – Simple algorithm extension

We have implemented a parallel variant of the basic algorithm that computes some of the alternative locations for the new vertex in parallel. Whilst this only generates a low level of concurrency, it can accelerate the search process between 2 and 3 times. When combined with multiple starts the algorithm can make effective use of a larger number of processors.

Divide-and-conquer

This method subdivides the domain, evaluates the cost at the subdivision points and uses these values to select a subset of the domain. This sub-domain becomes the new domain for the next iteration of this procedure. The process continues until the variation of cost over the subdivision points is within certain bounds.

The case of a one dimensional search space is simp lest to describe. Let $[a,b]$ be the original domain. We subdivide this interval into M steps using $M+1$ equally spaced points $a = x_0, x_1, ..., x_M = b$ and evaluate the cost at these points. Suppose that the minimum cost occurs at point x_i. Assuming this is not one of the end points, we select x_{i-1} and x_{i+1} as "brackets" for the solution. The minimum sought is assumed to be somewhere between these values. Then a is replaced by x_{i-1} and b by x_{i+1} and the process repeated. In the case where the minimum point x_i is equal to a then the brackets are taken as a and x_1 . In the case where the minimum point x_i is equal to b then the brackets are taken as x_{N-1} and b . The method is known to find the global minimum if the cost function is "unimodular". In most cases however the method should be considered as a heuristic that should find a local minimum.

The "Fibonacci search" method is similar to the method implemented; it uses $M = 3$ steps and points that are *not* equally spaced. Fibonacci search is known to be the most efficient possible in terms of the number of function evaluations. The Fibonacci method is not currently implemented in Nimrod/O.

For the case of an M dimensional search space, the domain for each dimension is subdivided. The number of steps used may vary between dimensions. For the *j*th dimension, suppose that the bounds are a_j , b_j and that M_j steps are used. This gives $M_j + 1$ values for this dimension. Taking all possible combinations of one value from each dimension yeilds a total of $(M_1 + 1)(M_2 + 1)...(M_N + 1)$ points, the "current grid". These are sent for parallel evaluation. The point that gives the minimum cost is the centre of the next iteration. Bracket are taken on either side for each dimension as for the one-dimensional case. These bracket define the subdomain for the next iteration.

A parallel version of the algorithm has been implemented. This variant evaluates the points of the domain in parallel, and thus the concurrency depends on the number of points in a domain. This can be quite high, and when combined with multiple starts, can utilise a large number of processors.

Simulated Annealing

Simulated annealing combines a downhill search with a random search. Suppose that initially we have a point x in the search space and that the cost at that point is f(x). A new point x' is randomly generated that is "nearby" in some sense; we will call this a "trial point". The cost there is f(x'). Next we decide whether to move to x', that is whether to replace x by x' as the current approximation. If f(x') < f(x) then the move is definitely accepted. If f(x') ≥ f(x) then the move is accepted with a probability of

$$\Pr(move\_accepted) = \exp\left(\frac{f(x) - f(x')}{T}\right)$$

Here T is a positive number called the “temperature”. If T is large then this formula gives a value close to 1. If T is small compared to f(x)-f(x'), then this number is very small. In summary, downhill moves are always accepted,

uphill moves may be accepted depending on the size of the change in cost relative to the temperature.

Once the move has been accepted or rejected, the process is repeated with a new trial point. This gives a sequence of current approximations, the "Markov chain" of points. Initially the temperature is set high so that almost all moves are accepted. Gradually the temperature is reduced; when it is low only downhill moves are accepted and the current approximation settles into a local minimum. Simulated annealing is recommended [26] over a pure downhill search when the landscape is rough as the uphill moves at high temperature provide a chance of escaping from a local minimum.

The Nimrod/O parallel implementation provides concurrency in three ways. First multiple starts are run in parallel. Secondly, all evaluations are cached avoiding repetition of the same evaluation. This is particularly important for simulated annealing as trial points may be revisited many times. Thirdly, the implementation uses and experimental parallelisation technique based on "speculative computing". This technique searches a tree of possible moves and evaluates the potential points in a batch.

Nimrod/O Architecture

Figure 2 shows a schematic of the Nimrod/O architecture. Nimod/O accepts a superset of the declarative “plan” files that are used to drive Nimrod, as discussed in [13]. In Nimrod/O additional statements are included that describe the optimization process that is to be used. Figure 3 shows and example of such a plan file, highlighting some of the new statements in Nimrod/O. The `parameter` and `task` statements define the optimisation variables, and the commands required for running the application. The `method` statements define various parameters for the search methods, such as how many concurrent starts are required, the tolerance, etc.

Nimrod/O has been built to support an extensible range of optimization procedures. Each of these procedures requires the evaluation of an objective function in order to proceed. This is performed by a request to the Nimrod/G or EnFuzion remote job execution engines. The algorithm forms a set of parameter values and passes these to Nimrod/G or EnFuzion for evaluation against the model. The model is run on an appropriate platform and the objective function value is extracted from the model output. A cache is superimposed between Nimrod/O and the backend to reduce the number of calculations required if the same parameter values are requested more then once. A persistent database is attached to the cache to support restart if Nimrod/O is terminated prematurely. By storing all function values, the user can restart the system from scratch and proceed to the same position without rerunning the computational models, providing a recovery process in the event of machine or network failure.

Nimrod/G and EnFuzion share a common API, and thus it is possible to execute the Nimrod/O model computations either on a local cluster, or on the Grid, depending on the available resources. This choice is transparent to the algorithms in Nimrod/O, and the selection of backend can be left to the user depending on the available resources and

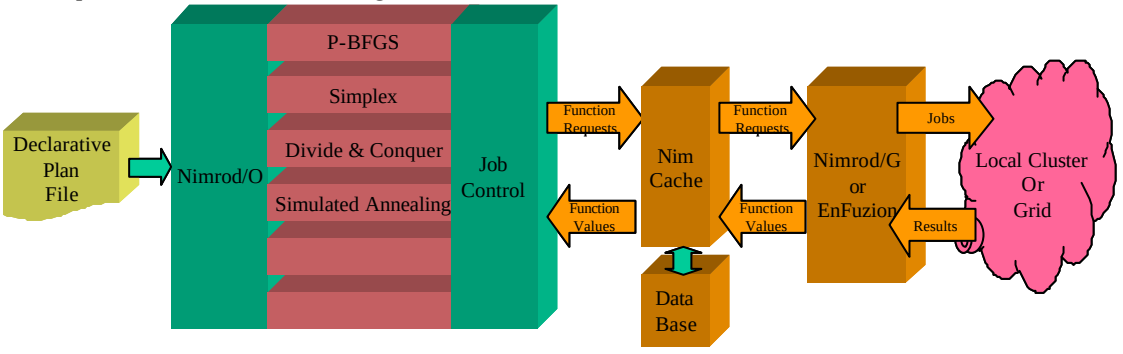


Figure 2 - Architecture of Nimrod/O

the number of processors required. Accordingly, scheduling of these jobs is also left to the backend technology. Figure 4 shows the Nimrod/G and EnFuzion dispatchers in use. In the Nimrod screen on the left, the coloured boxes represent the different instances of the model, and these can be seen assigned to hosts on the Grid. In this particular example Nimrod is supporting three different Grid middleware services, namely, Legion[15], Globus [16] and a Condor [17]. The EnFuzion screen dump shows the different model instances as they run on the various nodes of the local cluster. The differences between Nimrod/G and EnFuzion are discussed in other papers [18].

```
parameter x float range from -20 to 20
parameter y float range from 0 to 40
parameter z text select anyof "Red"
"White" "Blue"

task main
   copy  root:~/projs/* node:.
   node:execute ./run.script $x $y $z >
        final.objfn
   copy  node:final.objfn output.$jobname
endtask

method simplex
   starts 5
      starting points widespaced 3 5
      tolerance 0.10
   endstarts
endmethod

method bfgs
   starts 5
      starting points random
      tolerance 0.10
   endstarts
endmethod
```

Nimrod commands

Nimrod/O specific commands

Figure 3 - A sample Nimrod/O plan file.

An Aerofoil Case Study

Previously, we have applied Nimrod/O to a number of different problems, ranging from air pollution studies, computational electromagnetics and stress analysis [13]. The case study reported in this paper concerns the design of a simple aerofoil. Initially, the problem was developed and solved *without* Nimrod/O at the Centre for Advanced Numerical Computation in Engineering & Science (CANCES). Specifically, a simple two-dimensional aerofoil was modelled using a FLUENT simulation [19]. The shape of the simulation mesh was generated from the problem input parameters, and FLUENT was used to compute the flowfield around the aerofoil, from which lift and drag properties were derived. In addition, a special Fortran program was written to take the solutions from the FLUENT simulation and iteratively search for an optimal wing design using a Simplex method [20]. The aerofoil mesh generated by GAMBIT, had 28089 nodes and 49426 elements, made up of 43090 triangular elements and 6336 quad elements. The convergence criteria for the FLUENT run were set at 1.0e-4. The original purpose of the experiments was to investigate the applicability of optimization to the design of an aerofoil, with the goal of maximising the ratio of lift to drag. The solution took some time to develop because it required the development of code specifically tailored to the problem.

The aerofoil shape and configuration is governed by three parameters - the angle of attack, the camber and the thickness, as shown in Figure 5. At a particular configuration of these parameters, the objective function value, the ratio of lift to drag, is maximised. Complete enumeration of the space is infeasible because the number of simulations required is excessive.

Following the earlier work performed at CANCES, we applied the Nimrod/O tool to the same problem. Again, the original FLUENT code was used to model the aerofoil, however, rather than using the Fortran code developed specifically for the problem, Nimrod/O was used to perform the search. The study was an outstanding success in three ways:

- It only took about an hour to set up Nimrod/O on the Aerofoil problem.
- The results that were generated by Nimrod/O were better than those of the specific code.
- We were able to explore two algorithms by changing only the plan file

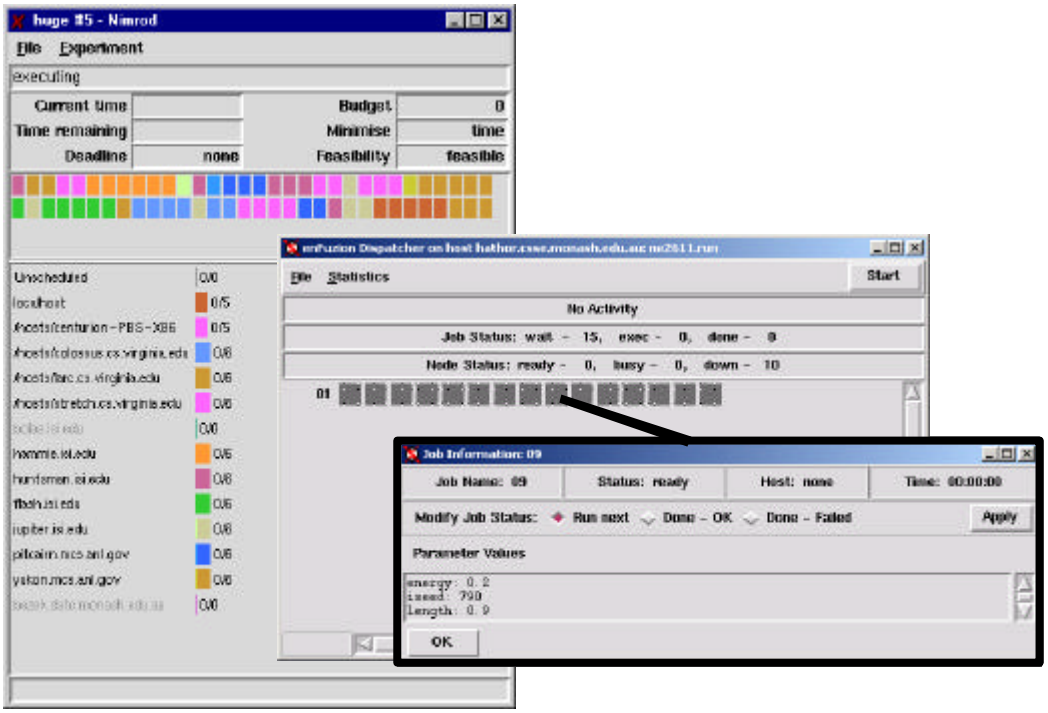


Figure 4 – Nimrod and EnFuzion engines

Table 1 summarises the results of the work. The "Best Objective Function" column shows the highest value of the lift/drag ratio achieved by the algorithm. The "Number of function evaluations" is the number of times the FLUENT simulation was executed. These simulations are much more expensive than the optimisation process itself. Therefore, the bulk of the run time is spent in executing the CFD code rather than performing the optimisation calculations. The table shows that our implementation of Simplex and P-BFGS both returned a better objective function values than the tailored Fortran code, but they required more function evaluations. Further, Simplex performed better than P-BFGS, and returned a value of 71.9 with fewer evaluations. The "Wall Clock Time" shows how long the Nimrod/O computations took on the VPAC AlphaServer SC[1], consisting of Alpha EV68's running at 833 MHz. The Tailored Fortran code ran on a different system, so the "Wall Clock Time" is unknown for this method.

[1] 32 Compaq ES40's (4 processors each) connected by a Quadrics interconnect.

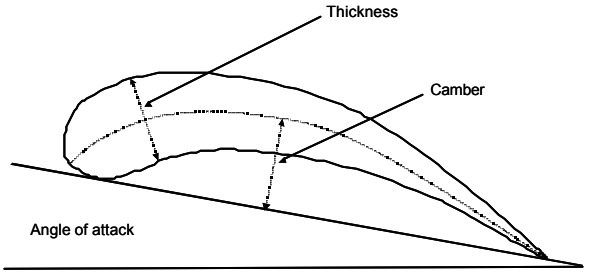


Figure 5 - Aerofoil structure

Table 2 shows the behaviour of the two Nimrod/O algorithms across multiple concurrent searches. Here we see the best, worst and average objective function values, and the number of function evaluations required to achieve them. Interestingly, the average result from the Nimrod/O Simplex is better than the tailored code, and achieved this with fewer function evaluations on average. However, the best P-BFGS still performed better than the average Simplex, but used more function evaluations.

Whilst the P-BFGS algorithm did not perform as well as Simplex, on average, on this particular problem, it is not possible to know this beforehand. *One of the key advantages of Nimrod/O over the tailored solution is that it is possible to try a number of different algorithms on the one problem.*

Figure 6 shows the tracks followed by the Simplex (the black curve) and the P-BFGS (the blue curve) algorithms in the three dimensional space. We have drawn three different iso-surfaces of the objective function, coloured yellow, green and red. The red iso-surface corresponds to an objective function value of 60, green corresponds to 52 and the yellow one corresponds to 47. Thus, in both cases, the tracks show the algorithms iteratively improving the solution until they terminate near the global optimum. Whilst both searches began at similar locations, they followed different paths and converged to different local minima, with Simplex yielding a better result. An interesting observation is that both of the Nimrod/O algorithms that were used yielded good solutions without requiring multiple starting locations. This is presumably due to the relatively smooth nature of the objective function. A movie form of Figure 4 is available on the web at [22]. Figure 7 shows the search paths taken by the 8 Simplex searches each from a different starting location. Whilst they all start at different locations, those that find good solutions converge on the same region of space, surrounded by the red iso-surface. A few of the searches terminate early, some distance from the global minimum.

Figure 8 shows the parallel behaviour of the system running 8 Simplex and 8 P-BFGS searches concurrently. Each of the search methods was run independently on a Beowulf cluster using EnFuzion as the backend technology. Since the machine size was limited to 64 processors, each of the different methods was run independently (as 2 lots of 8 concurrent starts) and then these two runs were added together and graphed. Whilst the current experiment used a local cluster, we have run similar experiments on the Grid using the Nimrod/G engine. Although the concurrency within the search algorithms is relative low, running multiple searches in parallel improves the resource utilization and also solves a given number of runs in less time. Further, we continue our search for efficient search algorithms that have more concurrency.

Conclusion

In this paper we have described a new design tool called Nimrod/O, and have demonstrated its effectiveness on solving a simple, but realistic, CFD problem. In comparison with an existing tailored simplex search, Nimrod/O allowed us to explore two different optimisation algorithms very quickly, and without any code modification. Both of our algorithms out-performed the tailored code, yielding better results in less time. Some low level concurrency in the algorithms further accelerated their performance. The Nimrod/O design philosophy allowed us to perform automatic design without tuning or modifying the existing FLUENT simulation. *The most dramatic result of the work was the ease of application – it only took us about an hour to set up Nimrod/O for a new optimisation problem.*

Method	Algorithm	Best Objective Function	Angle of Attack	Thickness	Camber	Number Function Evaluations	Wall Clock Time (hh:mm)
Tailored Fortran	Simplex	52.8	0.8	0.03	0.10	67	Unknown
Nimrod/O	Simplex	71.9	1.00	0.02	0.10	82	29:21
Nimrod/O	P-BFGS	69.0	1.11	0.03	0.11	113	43:40

Table 1 - Case study results

Method	Algorithm	Best		Worst		Average	
		Objective Function	Number Function Evaluations	Objective Function	Number Function Evaluations	Objective Function	Number Function Evaluations
Nimrod/O	Simplex	71.9	82	49.0	25	65.8	49.0
Nimrod/O	P-BFGS	69.0	113	44.8	99	54.6	76.1

Table 2 – Variance in algorithm behaviour

Clearly our work is in its early stages – the case study shown here is only a very simple design. In a real world engineering environment, one would expect the computational models to be much more complex, involving multiple simulation techniques. Also, an industrial strength problem would have more parameters. How well our work scales can only be determined by experimentation. However, the early results reported here and in [13] are extremely encouraging.

The choice of back end technology allows us to abstract the exact nature of the computational platforms, and thus the user is free to utilise machines ranging from a single high-end workstation through to the computational Grid. For small studies a local cluster provides sufficient performance. However, when multiple searches are required, using different algorithms, the number of concurrent executions exceeds the resources available on a local cluster, and the Grid becomes a viable platform. We expect to perform some larger design experiments in the area of mechanical durability in the near future.

Acknowledgments

This work has been supported by the Australian Research Council (ARC), the Victorian Partnership for Advanced Computing (VPAC) and the Centre for Advanced Numerical Computation in Engineering & Science (CANCES). The programs were run on facilities at CANCES and VPAC. We would like to acknowledge Ivan Mayer for his assistance, and Jimmy Li for his work on the original optimization code.

References

[1] Abramson D., Sosic R., Giddy J. and Hall B., "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[2] Abramson, D. “Parametric Modelling: Killer Apps for Linux Clusters”, The Linux Journal, #73, May 2000, pp 84 – 91

[3] Lewis, A., Abramson D., Sosic R., Giddy J., “Tool-based Parameterisation : An Application Perspective”, Computational Techniques and Applications Conference, Melbourne, July 1995.

[4] Foster, I. and Kesselman, C. Globus: “The Grid: Blueprint for a Future Computing Infrastructure”. Morgan Kaufmann Publishers, 1999.

[5] Eldred, M., Outka, D., Fulcher, C. and Bohnhoff, W. “Optimization of complex mechanics simulations with object-oriented software design.”, Proceedings of the 36th IAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, pp 2406 - 2415, New Orleans, LA, April 1995

[6] Czyzyk, J, Owen, J. and Wright, S. “Optimization on the Internet”, OR/MS Today, October 1997.

[7] Casanova, H. and Dongarra, J. “NetSolve: A Network Server for Solving Computational Science Problems”, The International Journal of Supercomputing

Applications and High Performance Computing, Vol 11, Number 3, pp 212-223, 1997.

[8] Parker, S. and Johnson, C. "SCIRun: A Scientific Programming Environment for Computational Steering", Proceedings of IEEE Supercomputing 1995, San Diego, December 95.

[9] McCorquodale, J., Parker, S., Davison de St. Germain, J. and Johnson, C., "The Utah Parallelism Infrastructure: A Performance Evaluation", 2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference, April 22-26, 2001, Seattle, Washington (USA).

[10] Taha, H., (1992) "Operations Research: An introduction" Macmillan Publishing Company, New York, 5th Ed, 822 pages.

[11] Press. W. H., "Numerical Recipes: the Art of Scientific Computing". Cambridge University Press, 1986.

[12] Gill, P.E., Murray, W. and Wright. M, "Practical Optimization". Academic Press, London, 1981.

[13] Abramson, D, Lewis, A. and Peachy, T., "Nimrod/O: A Tool for Automatic Design Optimization", The 4th International Conference on Algorithms & Architectures for Parallel Processing (ICA3PP 2000), Hong Kong, 11 - 13 December 2000.

[14] Abramson, D, Lewis, A. and Peachy, T. "Case Studies in Automatic Design Optimisation using the P-BFGS Algorithm", 2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference, April 22-26, 2001, pp 104 – 109, Seattle, Washington (USA).

[15] Grimshaw, A. and Wulf, W., et al, "The Legion vision of a woroldwide virtual computer", Communication of the ACM, 40(1):39-45, 1997.

[16] Foster, I., Kesselman, C., *Globus: A Metacomputing Infrastructure Toolkit*, International Journal of Supercomputer Applications, 11(2):115-128, 1997.

[17] Litzkow, M., Livny, M. and Mutka, M.W., "Condor – A Hunter of Idle Workstations", proceedings of the 8th International Conference of Distributed Computing Systems, pp 104-111, June 1988.

[18] Abramson, D., Giddy, J. and Kotler, L. "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", International Parallel and Distributed Processing Symposium (IPDPS), pp 520- 528, Cancun, Mexico, May 2000.

[19] http://www.fluent.com/

[20] Nelder, J.A., and Mead, R.,"A simplex method for function minimization", Comput. J., 7, pp. 308-313, 1965.

[21] Fletcher. R. "Practical Methods of Optimization." 2nd ed., Wiley, New York, 1987.

[22] http://www.csse.monash.edu.au/~davida/nimrodo/aerofoil.mpg

[23] van Laarhoven, P.,Aarts E., (1987): "Simulated Annealing: Theory and Applications" D Reidel Publishing Company, Dordecht, 186 pages.

[24] Poloni, G and Pediroda, V. "GA Coupled with Computationally Expensive Simulations: Tools to Improve Efficiency", Chapter 13, in "Genetic Algortihms and Evolution Strategy in Engineering and Computer Science", Edited by Quagliarella, Periaux, Poloni and Winter, John Wiley & Sons, 1998, ISBN 0 471 97710 1.

[25] http://www.lmsintl.com/

[26] van Laarhoven P.J.M. and E.H.L. Aarts, Simulated Annealing: Theory and Applications, Reidel, 1987.

[27] Spendley, W, Hext, G. R. and Himsworth, F. R. "Sequential applications of simplex designs in optimization and evolutionary operation", Technometrics, 4, pp 441-461, 1962.

Surface	Objective Function Value
Red	60
Green	52
Yellow	47

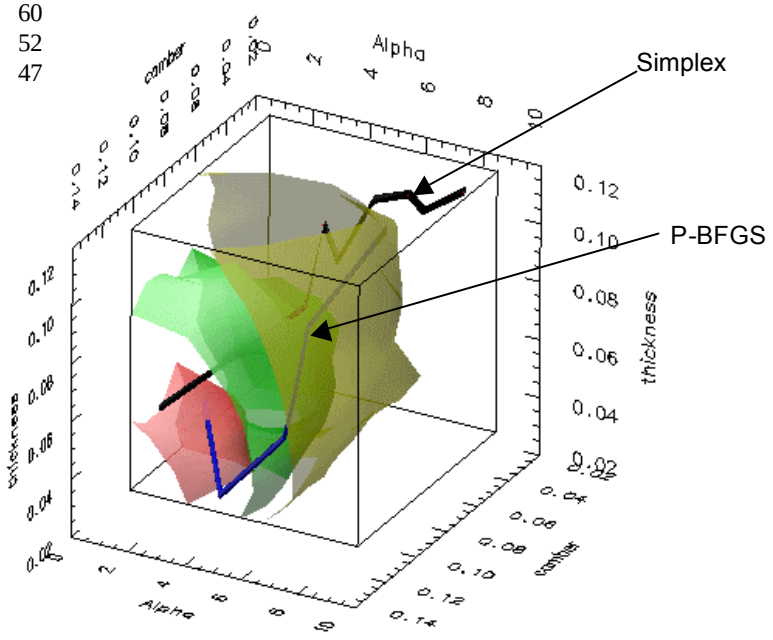


Figure 6– Simplex and P-BFGS search tracks

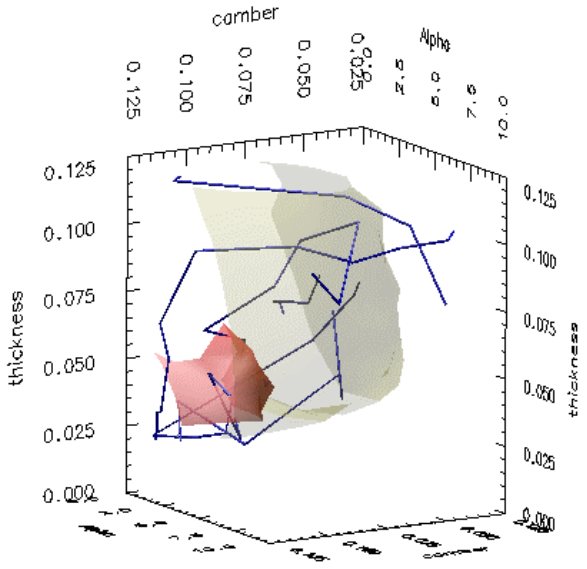


Figure 7 – 8 Simplex search tracks

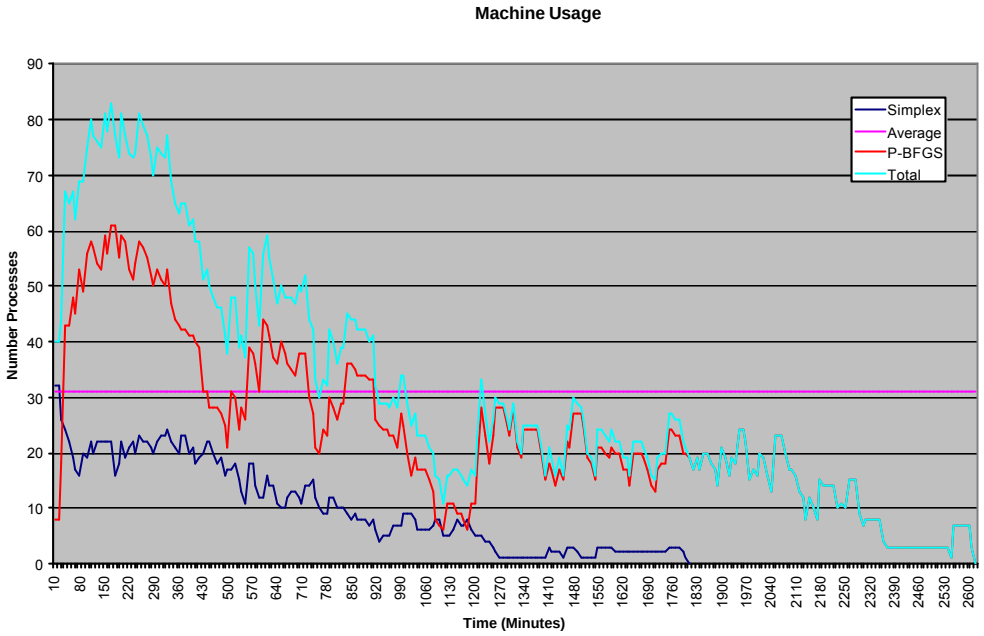


Figure 8 – Parallel Behaviour of multiple searches.

NIMROD/O: A TOOL FOR AUTOMATIC DESIGN OPTIMISATION USING PARALLEL AND DISTRIBUTED SYSTEMS

DAVID ABRAMSON †, ANDREW LEWIS §, TOM PEACHEY †

† School of Computer Science and Software Engineering, Monash University, CLAYTON, VIC 3168, Australia
davida@csse.monash.edu.au

§ High Performance Computing Facility, Griffith University, Nathan, Qld, 4111, Australia

This paper describes a novel tool called Nimrod/O that allows a user to run an arbitrary computational model as the core of a non-linear optimization process. Nimrod/O allows a user to specify the domain and type of parameters to the model, and also a specification of which output variable is to be minimized or maximized. Accordingly, a user can formulate a question like: "what parameter settings will minimize the model output**?**'. Nimrod/O currently employs a number of built-in optimization algorithms, namely BFGS, Simplex, Divide and Conquer and Simulated Annealing. Jobs can be executed on a variety of platforms, including distributed clusters and Computational Grid resources. The paper demonstrates the utility of the system with a number of case studies.

1 Introduction

Computational science and engineering (CS&E) has become the preferred method of design for a number of disciplines, including aircraft and automotive manufacture, electronics and RF design, environmental engineering, chemical engineering and rational drug design. The driving technologies common to these different areas are the availability of affordable super-computing hardware and robust computational models. The most significant advantage of CS&E is that it allows a designer to experiment with a number of different scenarios before committing to an actual implementation, so the design cycle is reduced and the product (or outcome) quality can be improved.

To date, there are very few generic computational frameworks that support the routine application of CS&E in the product design and development cycle. Thus, in many cases software developers must build an appropriate computational model using conventional programming languages, and then run the model as though it were just another executable program. Whilst this is sufficient, it can be error prone when a large number of alternatives are being considered, because the user must track the path taken as variables are changed and designs are evaluated. Further, if any form of *automatic* optimization is required (like minimizing or maximizing some objective function), then the optimization code has to be integrated into the

computational model itself. The major disadvantage of this approach is that the optimization technology must be *re-engineered* each time a new model is applied.

For a number of years we have been experimenting with a generic tool, called Nimrod, that allows a user to execute a CS&E model over a range of design parameters using high performance computers. Nimrod makes it very easy for a user to specify a range of parameters over which a model must be executed. It then distributes each different execution across a range of computational resources, such as individual workstations, parallel machines and clusters. A version of Nimrod called Nimrod/G supports execution on a computational Grid, in particular, using the Globus toolkit [1][2][3]. A new version called Nimrod/L is currently being built using the Legion environment [4], and a commercial version called enFuzion is also available for workstation clusters [5][6]. A wide range of case studies has been developed [7][8][9][10].

Nimrod can be used to support CS&E based design because it allows a user to explore many different scenarios selecting the ones that minimize or maximize the objective function. However, its biggest disadvantage is that Nimrod explores *all of the search space exhaustively*. Whilst this is feasible for small experiments it rapidly becomes impractical for large, high dimensioned problems.

A number of projects have addressed parts of the design optimization problem over the years. The DAKOTA project at Sandia laboratories [11] investigated the combination of optimization with computational models. DAKOTA is a C++ based tool kit that allows a user to choose different optimization algorithms to guide the search for optimal parameter settings. DAKOTA has been successfully demonstrated on a number of structural mechanics applications. However, DAKOTA does not support rapid prototyping and still requires the construction of new programs for each different type of experiment which is performed. NEOS is a distributed web based optimization engine that allows a user to solve optimization problems using special remote optimization servers. However, NEOS relies on the objective function being specified in an algebraic form and does not support objective functions implemented by CS&E models. Likewise, NetSolve [12] provides a web based engine for solving linear algebra problems, but does not explicitly address optimization using CS&E objective functions. SciRun [13] is an interactive tool which allows a user to build a CS&E model very rapidly using a graphical programming interface. However, it does not support optimization.

In this paper we describe Nimrod/O, a tool which augments the distributed parameter sweep functionality of Nimrod with automatic optimization. We believe that Nimrod/O is the first system to combine rapid application development, parallel and distributed supercomputing and optimization in the one tool. Nimrod/O is modular, and incorporates a number of standard non-linear optimization methods. It uses a declarative style developed for Nimrod, which allows a user to specify the types of parameters and commands necessary for running the underlying CS&E model. The user can then specify which optimization algorithm (or combination of

algorithms) to apply to the problem. Thus, if a user already has a robust CS&E model, it can be converted into a parallel optimizing design tool very quickly.

The paper describes the Nimrod system as background, and then shows the architecture of Nimrod/O. It then discusses the types of optimization algorithms that are supported, followed by a set of case studies. The case studies implement problems in environmental engineering, mechanical engineering and RF electronic design.

2 Nimrod and Nimrod/G

Nimrod is a tool that manages the execution of parametric studies across distributed computers. It takes responsibility for the overall management of an experiment, as well as the low-level issues of distributing files to remote systems, performing the remote computation and gathering the results. EnFuzion [5] is a commercial version of the research system Nimrod.

When a user describes an experiment to Nimrod, they develop a *declarative* "plan" file which describes the parameters, their default values, and the commands necessary for performing the work. The system then uses this information to transport the necessary files and schedule the work on the first available machine.

When the user invokes Nimrod on a workstation, the machine becomes known as the "root" machine because it controls the experiment. When the dispatcher executes code on remote platforms, each of these is known as a computational "node". Thus a given experiment can be conducted with one root and multiple nodes, each of a different architecture if required.

Nimrod/G is a "Grid aware" version of Nimrod built on the Globus toolkit [1]. It exploits an understanding of its problem domain as well as the nature of the computational Grid to provide a high level interface to the user. Specifically, it provides transparent access to the computational resources, and implements user level scheduling. Nimrod/G is described in more detail in [26] and [27].

Nimrod/O makes use of Nimrod to perform its remote execution, as discussed in the next section. This isolates the optimisation algorithms from the functions required to access high performance computers of varying architecture.

3 Nimrod/O architecture

The architecture of Nimrod/O is summarized in Figure 1. Nimrod/O accepts a declarative plan file as previously developed for Nimrod. This file indicates the name and domains of the various parameters and provides descriptions of how to run the CS&E model.

Nimrod/O is built to allow the integration of a number of different optimization engines. These are linked together into the one executable, and are chosen by command line arguments. The CS&E model is actually executed on a distributed

supercomputer using Nimrod, so Nimrod/O is not concerned with the details of how jobs are run. Since Nimrod, Nimrod/G and enFuzion all share a common API, it is possible to substitute different back end engines depending on the type of supercomputing facilities that are available. For example, the Nimrod/G backend allows jobs to be run on a globally distributed computation grid like GUSTO [2][16], whereas the enFuzion backend supports department level clusters efficiently.

The system supports concurrency in a number of places. First, if the algorithm itself is parallel (like P-BFGS discussed in the next section) then the system supports concurrent execution within the algorithm. For example, in P-BFGS a number of points are evaluated concurrently, and Nimrod/O does this by requesting Nimrod to run the model multiple times with different parameters. Second, the system allows different searches (possible using different optimization algorithms) to be performed in parallel using different starting locations. Because many of the algorithms we have considered depend on the choice of an initial point, it is important to run the optimization more than once, starting it at different places. Nimrod/O supports this by multi-threading its internal control and using the Nimrod backend to run multiple jobs. When we evaluate the performance of the system later in the paper, we use this feature to demonstrate the variability in final objective function values that are obtained.

Currently, the value of the objective function is returned from the application via a special file. However, in future we expect to add other methods of extracting the function value such as using environment variables to convey the information. A special cache server (called NimCache) is placed between Nimrod/O and Nimrod. This process caches all function requests, and if it detects a request for a point that has already been calculated, it returns the previous result.

4 Optimization algorithms

The current implementation of Nimrod/O supports 4 non-linear search heuristics, namely P-BFGS, Simplex, Divide and Conquer and Simulated Annealing.

4.1 P-BFGS

In general, gradient descent methods like the BFGS method, use the derivative of the cost function, as well as its value, to select a search direction, essentially reducing the multivariate optimization problem to uni-variate minimization along a search vector; a line search. They thus consist of two main operations that are executed repeatedly, namely gradient calculations and line searching. The P-BFGS algorithm is based on the quasi-Newton BFGS (Broyden, Fletcher, Goldfarb, Shanno) method, widely regarded as one of the most efficient and robust gradient descent methods for use with continuous functions [17][18][19].

The BFGS method maintains an approximation to the Hessian, *H*, of *f*, where: New approximations to the solution vector, *x*, are derived by:

1. Compute a search direction, $d = -H^{-1}\nabla f(x)$
2. Find a new *x*, using a line search, i.e. $x_+ = x + \lambda d$
3. Update *H* using the current approximation to *H*, *x* and x_+ $H_{ij} = \partial^2 f / \partial x_i \partial x_j$

The P-BFGS algorithm performs step 1 by computing the finite difference approximations for the gradient in parallel. It also uses a modified parallel line search algorithm for step 2. There is insufficient space in this paper to discuss the algorithm in detail, however, more information can be found in [20].

4.2 Simplex

The Simplex algorithm used is based on the method of Nelder and Mead [21]. In common with other such methods, at each iteration it examines a simplex of points and then changes the simplex in response. A simplex is a geometrical figure consisting, in N dimensions, of N+1 points, or vertices, and all their interconnecting line segments. For example, in two dimensions, a simplex is a triangle.

The algorithm is started with N+1 points, defining a simplex, and their associated cost function evaluations. It then takes a series of steps, at each step moving the point of the simplex where the cost function is highest through the opposite face of the simplex to a lower point.

If the cost function value at a reflection trial point is found to be particularly advantageous, the simplex is expanded in the given direction; if it is less advantageous, the simplex is contracted. Should no progress be possible, the simplex is simultaneously contracted in all dimensions. These steps are repeated until the cost function values at all points fall within a desired tolerance of each other.

The basic method has been modified in the current study in two ways.

- Simple bounds have been applied to all trial points, so that the search is contained within a given space.
- The evaluation of trial points has been parallelised. At each step, in a high proportion of cases at least two function evaluations will be made: the initial reflection, and either an expansion of it, or a contraction. The modified algorithm evaluates all three of these possibilities concurrently, yielding a speedup observed to approach a factor of two, for the use of three processors.

4.3 Divide and Conquer

The Divide and Conquer (DC) method is a simple heuristic which performs a repeated subdivision of the search space. The domain for each of the N parameters is subdivided into three parts and the objective function is computed at the $(3+1)^N$ grid points that result. The grid point which produces the minimum objective and its neighbours give the domain for the next iteration. There are two variants of this procedure, depending depending on a parameter called "drift". If "drift" is allowed then the next search is centred on the end value, so the next search domain will be partly outside the previous space. Alternately with drift disabled the new domain will be a proper subset of the previous space. The search terminates when the absolute value of the difference in the objective values at the best and worst grid points, as a proportion of the mean of those values, is less than the tolerance.

4.4 Simulated Annealing

Simulated annealing (SA) is a *Monte Carlo* method, based on the analogy between the aggregation of many particles in a physical system as it is cooled and the optimization of large combinatorial problem. In the physical system, annealing is the process of heating a solid until it melts, followed by a cooling operation until the substance crystallizes. Fast cooling leads to a process called *quenching* in which the crystal that is formed has a metastable amorphous structure (or imperfect lattice), with a very large internal energy. Slow cooling allows the system to reach its equilibrium (lowest energy level) at each decremental temperature. Hence it is possible to reach the zero energy level or ground state. In the optimisation perspective, search space can be viewed as a physical system in which elements or variables correspond to particles. The atomic bonding force is equivalent to the set of constraints imposed on the problem. The energy of the system is modelled as the penalty cost, so that when the system reaches the ground state, the process arrives at the global optimum configuration. Simulated Annealing has been applied to a wide range of problems from continuous systems to combinatorial optimization. In this work, we have experimented with both a public domain package [22] and one that we have written ourselves.

5 Case Studies

In this section we examine the performance of Nimrod/O on 5 real world case studies, shown in Table 1. Table 2 shows which algorithms were applied to the case studies, since we were not able to apply all algorithms to all problems. The exact details of the case studies are not discussed here. All of them required the optimisation of a particular objective function value. The number of parameters is shown in Table 1, and varies from 1 to 7.

Problem	Number of Dimensions	Description
Smog	2	Photo Chemical Pollution Model [23]
S-Plate	1	Static Strength of a Biaxially Loaded Plate – Simple cutout
C-Plate	3	Static Strength in a Biaxially Loaded Plate – Complex cutout
Ceramic	3	Radio frequency properties of a ceramic bead [24]
Antenna	7	Multi frequency antenna

Table 1 – Problem definitions

Problem	Algorithm				
	Exhaust	P-BFGS	Simplex	DC	SA
Smog	X	X			
S-Plate	X	X			
C-Plate	X	X	X	X	
Ceramic	X	X	X		X
Antenna			X		

Table 2 – Problem-Algorithm assignment

5.1 Results

Table 3 shows the results obtained when an exhaustive search is performed on 4 of the case studies. In these cases, the domain of the parameters is either integer, or converted into a number of discrete values. We were only able to run an exhaustive search on 4 of the problems, because the search space for the Antenna problem is too large to enumerate, even for a coarse decomposition.

The "# Func. Evals" column indicates how many model executions were required. The "Optimal Cost" column shows the value of the best objective function value. The "Time for 1 Func. Eval" column shows the average time to run the model once. The "Number of CPUs" column shows how many processors were available on the cluster we used. The "Wall Clock Time" column shows how long it took to run the entire experiment with 64 CPUs.

From these results, it can be seen that for many of the problems, an exhaustive search is expensive even when a reasonably sized cluster is available. For example, the Smog case study required 60 days of continuous use of a 64 way cluster!

Table 4 shows the results of applying the various algorithms to the problems. The "#Starts" column indicates how many independent runs were performed for the problem-algorithm pair. The "Best Cost", "Worst Cost" and "Mean Cost" columns indicate the best, worst and mean costs across the independent starts. If only one start was performed then the worst and mean are omitted. If a '*' is present in the

"Opt" column, then the best cost was optimal. The "# Iterations" indicates how many iterations of the algorithm were required to achieve the best reported solution. For Simplex, DC and SA this involves sending out a number of function values for concurrent evaluation before computing a new position. In P-BFGS, each iteration involves both a gradient computation and a line search. The "#Func Evals" column reports the total number of times the model was run in order to find the best solution. The "Wall Clock Time" column indicates how long it took to find the best solution. The last column shows the ratio of the wall clock time for each problem with the time for an exhaustive search (from Table 3). The "Ave # of Processors Required" indicates how many processors were needed on average to achieve the best result in the time shown. This is calculated on the assumption that all independent starts are run concurrently and that each algorithm requires some number of processors to operate in the time that is reported.

One of the most striking features of the data in Table 4 is the variability of the results across different problems and search algorithms. P-BFGS performs well on all problems for which the optimal cost was known, achieving optimal in all cases. However, Simplex does not manage to achieve optimal on any of the problems attempted, although the comparison is a little unfair since the running times are quite different from P-BFGS. In addition, the cost achieved on C-Plate using Simplex, whilst not optimal, is also quite acceptable. DC performs well on the one problem on which it was attempted, and our particular version of SA does not perform well on Ceramic. However, this SA code is still under development and further enhancements may substantially improve its performance. Further, we obtained good results using the ASA package [22] on the Ceramic problem, but did not have all of the statistics to allow a fair comparison at this stage. Accordingly, we have not reported the ASA results in Table 4. Our results do indicate that one of the algorithms managed to achieve an optimal cost for each of the problems, validating the idea of making multiple algorithms available from the one optimization package.

The timing performance of the various algorithms indicates that whilst some of the searches might still take a large amount of time, in most cases the automatic optimization was faster than an enumerative search. For the Smog problem, P-BFGS achieves optimal 17.4 times faster than a 64 processor cluster performing exhaustive search. Moreover, it does this with only 5.1 processors on average. Similarly, P-BFGS achieves optimal on Ceramic 11.2 times faster than exhaustive search, using only 27.1 processors on average. At the other extreme, P-BFGS is about the same speed as an exhaustive search on C-Plate using 48 processors. However, DC solves this problem 5 times faster than exhaustive search with 53.8 processors. The variation in the number of processors used is a reflection of the parallelism available within the optimisation algorithm itself, and the number of parallel starts that were performed. For example, running Simplex on the Antenna problem did not use many processors because we only performed one start, and the algorithm has little internal parallelism. The DC algorithm on the C-Plate problem,

on the other hand, achieves high parallelism using only one start because the algorithm itself is naturally parallel.

Problem	# Func Evals	Optimal Cost	Time for 1 Func. Eval. (Secs)	Number of CPUs	Wall Clock Time (Hrs:Mins:Secs)
Smog	10000	68	31680	64	1375: 0: 0
S-Plate	201	15.0	204	64	0: 10: 40
C-Plate	2541	14.7	326	64	3: 35: 35
Ceramic	12896	-39.8	3000	64	167: 55: 0
Antenna	-	-	7800	-	Not Possible

Table 3 – Exhaustive Search of case studies

It may be noted, since the exhaustive search data was available for 4 of the problems, it was not necessary for search algorithms to actually run the model again in order to provide the timing results in Table 4. In particular, for the Ceramic problem, we allowed the algorithms to perform a table lookup using the data from the exhaustive search, and then we predicted the total run time using the time required to execute the model once. Thus, when SA was performed on the Ceramic problem, we did not actually have to wait 370 hours for the result.

6 Conclusions

In this paper we have shown the design and implementation of a novel tool called Nimrod/O, which allows a user to produce an optimizing decision support system using existing computational models. The tool provides a range of optimization algorithms, and using a simple declarative specification technique, these algorithms can be applied to an arbitrary computational model even if source code for the model is not available. A range of backend tools means that we can target platforms ranging from department level clusters to a global computational grid.

The paper reports work which is in progress, so we do not regard Nimrod/O as a completed system. We plan to perform more work on the core optimization algorithms to improve their performance. We also plan to improve the engineering of Nimrod/O to allow arbitrary external optimization libraries to be integrated easily. In the coming months we plan to add a restart and recovery feature to allow long running experiments to be terminated and restarted at a later time. One of the most significant algorithmic enhancements is to improve the parallelism of the optimization methods themselves, with the hope of delivering shorter run times overall.

Acknowledgements

The Nimrod/O project has been supported by the Australian Research Council, as part of an ARC Collaborative Grant. We wish to acknowledge our partners in this grant, namely the Environment Protection Authorities of Queensland, Victoria and Western Australia. We wish to thank Turbo Linux Inc. for access to the enFuzion software, and the Distributed Systems Technology CRC, for access to Nimrod/G. We also acknowledge our research partners in the Globus group, in particular Ian Foster from Argonne National Laboratories.

The case studies discussed in this paper involved the work of a number of colleagues. We wish to acknowledge

- Dr Martin Cope, Dr Steven Spencer and Dr Maciej Skierski for their assistance with the Smog study;
- Professor Rhys Jones, Dr John Sawyer and Mr Paul Chaperon for their assistance with the Plate study;
- Mr Seppo Saario and Dr Jun Lu for their assistance with the Ceramic and Antenna studies.

References

1. Foster, I., Kesselman, C., *Globus: A Metacomputing Infrastructure Toolkit*, International Journal of Supercomputer Applications, 11(2):115-128, 1997.
2. http://www.globus.org/
3. Foster, I. and Kesselman, C. Globus: A Toolkit-Based Grid Architecture. In The Grid: Blueprint for a Future Computing Infrastructure, pages 259-278. Morgan Kaufmann Publishers, 1999.
4. Grimshaw, A. and Wulf, W., et al, "The Legion vision of a woroldwide virtual computer", Communication of the ACM, 40(1):39-45, 1997.
5. Turbo Linux Inc, www.turbolinux.com
6. Abramson, D. "Parametric Modelling: Killer Apps for Linux Clusters", Linux Journal, May 2000.
7. http://hathor.csse.monash.edu.au/RJK/
8. Abramson, D., Power, K., and Sosic, R., "Simulating Computer Networks using Clusters of PCs", HPC-TelePar'99 at the 1999 Advanced Simulation Technologies Conference (ASTC'99), April 11-15, San Diego, California,USA
9. D. Abramson, I. Foster, J. Giddy, A. Lewis, R. Sosic, R. Sutherst, N. White, "The Nimrod Computational Workbench: A Case Study in Desktop Metacomputing", Australian Computer Science Conference (ACSC 97), Macquarie University, Sydney, Feb 1997, pp 17 - 26.

10. Lewis, A., Abramson D., Sosic R., Giddy J., “Tool-based Parameterisation : An Application Perspective”, Computational Techniques and Applications Conference, Melbourne, July 1995.
11. Eldred, M., Outka, D., Fulcher, C. and Bohnhoff, W. “Optimization of complex mechanics simulations with object-oriented software design.”, Proceedings of the 36th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, pp 2406 - 2415, New Orleans, LA, April 1995.
12. Casanova, H. and Dongarra, J. “NetSolve: A Network Server for Solving Computational Science Problems”, The International Journal of Supercomputing Applications and High Performance Computing, Vol 11, Number 3, pp 212-223, 1997.
13. Parker, S. and Johnson, C. “SCIRun: A Scientific Programming Environment for Computational Steering”, Proceedings of IEEE Supercomputing 1995, San Diego, December 95.
14. Abramson D., Sosic R., Giddy J. and Hall B., "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.
15. Foster, I., and Kesselman, C. (editors), *Computational Grids. The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann Publishers, USA, 1999.
16. http://www-fp.globus.org/overview/testbeds.html
17. Gill, P.E., Murray, W. and Wright. M, “Practical Optimization”. Academic Press, London, 1981.
18. Fletcher. R. “Practical Methods of Optimization.” 2nd ed., Wiley, New York, 1987.
19. Press. W. H. , “Numerical Recipes: the Art of Scientific Computing”. Cambridge University Press, 1986.
20. Lewis, A and Abramson, D. “A Parallel BFGS algorithm and its application to Automatic Design Optimization”, Griffith University High Performance Computing Facility Technical Report, 2000.
21. Nelder, J.A., and Mead, R.,"A simplex method for function minimization", Comput. J., 7, pp. 308-313, 1965.
22. Ingber, L, “Very fast simulated re-annealing”, Journal of Mathematical Computer Modeling, Vol 12, #8, pp 967-973, 1989. See http://www.ingber.com.
23. McRae G.J., Russell A.G. and Harley R.A., 1992. CIT photochemical airshed model -users manuals, Carnegie Mellon University, Pittsburgh, PA and California Institute of Technology, Pasadena, CA.
24. Lewis, A., Saario, S., Abramson, D. and Lu, J., “An Application of Optimisation for Passive RF Component Design”, to appear, Conference on Electromagnetic Field Computation, Milwaukee, June 4-7th, 2000.

25. Lewis, A, Abramson, D and Simpson, R., "Parallel non-linear optimization : Towards the design of a decision support system for air quality management", IEEE Supercomputing 97, San Jose, 1997.
26. Abramson, D., Giddy, J. and Kotler, L. "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", International Parallel and Distributed Processing Symposium (IPDPS), pp 520-528, Cancun, Mexico, May 2000.
27. Buyya, R., Abramson, D. and Giddy, J. "Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid", HPC Asia 2000, May 14-17, 2000, pp 283 – 289, Beijing, China.

Problem	Alg'm	Starts	Best Cost	Opt	Worst Cost	Mean Cost	# Iter'ns for Best	# Func Evals for Best	Wall Clock for Best (Hrs:Mins:Secs)	W'll Clk Exh' st / W'll Clk Best	Ave # of Procs req'd
Smog	P-BFGS		68	*	-	-	2	46	79: 12: 0	17.4	5.1
S-Plate	P-BFGS		15	*	15	15	2	46	0: 34: 0	0.3	36.8
C-Plate	P-BFGS		14.7	*	15.7	15.1	7	240	3: 37: 11	1.0	48.0
C-Plate	Simplex		15		16.7	15.4	7	21	0: 38: 0	5.7	24.0
C-Plate	DC		14.7	*	-	-	8	430	0: 43: 26	5.0	53.8
Ceramic	P-BFGS		-39.8	*	26.3	-2.1	5	61	15: 0: 0	11.2	27.1
Ceramic	Simplex		-9.4		37.4	16.9	10	24	9: 10: 0	18.3	17.5
Ceramic	SA		-18.5		-14.2	-17.5	22	472	370: 0: 0	0.5	8.5
Antenna	Simplex		35.7	-	-	-	23	84	49: 50: 0	-	3.7

Table 4 – Results of Case Studies with Nimrod/O

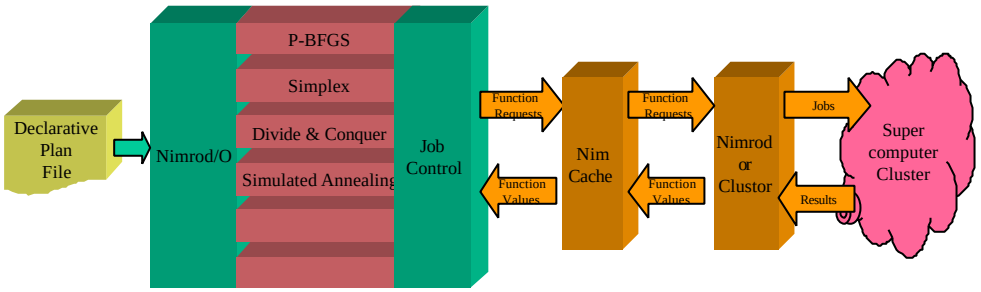


Figure 1 - Architecture of Nimrod/O

Optimization using Nimrod/O and its Application to Robust Mechanical Design

Tom Peachey[1], David Abramson[1], Andrew Lewis[2], Donny Kurniawan[1] and Rhys Jones[3]

[1] School of Computer Science and Software Engineering, Monash University, Melbourne, Australia
[2] Div. of Information Services, Griffith University, Brisbane, Australia
[3] Department of Mechanical Engineering, Monash University, Melbourne, Australia
tcp@csse.monash.edu.au

Abstract. We describe the Nimrod/O distributed optimization tool and its application to a problem in mechanical design. The problem is to determine the shape for a hole in a thin plate under load that gives optimal life in the presence of flaws. The experiment reveals two distinct design strategies for optimizing this life. Nimrod/O is able to find both of these rapidly due to its inherent parallelism.

1 Nimrod/O

Advances in computing science and engineering modeling have enabled design engineers to use computational models instead of real world prototypes in many situations. Such an approach is usually faster and cheaper and hence allows the user to explore various design scenarios. In particular the user may search through combinations of design parameters in order to achieve a design that is optimal in some sense. Similarly, scientific research is increasingly using computational models. Often the aim is to determine those model parameters that produce the best fit to real world data, so again the model is used repeatedly to achieve some optimum.

Such models typically require computational times of minutes or hours, even on a high end processor. We have produced a number of tools that facilitate concurrent execution of these models. The tools Nimrod and EnFuzion [1] allow the user to specify a range of values for each design parameter; then all combinations of these values are generated and the resulting jobs are farmed to a cluster of processors. Nimrod/G [2] is a version of Nimrod that utilises resources on a global computational grid. These tools may be used for design optimization by exploring all possibilities in a search space. A more efficient search is provided by Nimrod/O [3],[4],[5] which offers a variety of standard optimization procedures to explore the parameter space.

When a Nimrod/O procedure requires values of the objective function it passes the parameters to Nimrod or EnFuzion for evaluation against the computational model. The model is run on an appropriate platform and the objective

function value is extracted from the model output. A cache is superimposed between Nimrod/O and the back-end to reduce the number of calculations required if the same parameter values are requested more than once. A persistent database is attached to the cache to support restart if Nimrod/O is terminated prematurely. The Nimrod/O implementation of the optimization procedures uses concurrent evaluations where possible. In addition it allows separate optimizations to be run concurrently.

To perform a Nimrod/O optimization the user needs to prepare a simple text "schedule" file. This specifies the parameters of the search space, any constraints imposed on the solution, how to run the computational model to obtain objective values and the optimization method(s) to apply. Assuming the model correctly executes on the remote nodes, no further development is required.

2 Computation of Fatigue Life for Shape Optimization

In this paper we discuss the application of Nimrod/O to the optimal design of mechanical components. The failure of such components is usually due to slow growth of a pre-existing crack followed by a sudden fracture [6]. A common mechanism for crack growth is fatigue due to cyclical loading. As a load is applied the high stress at the crack tip causes plastic deformation which produces an irreversible growth in the crack. This growth is small (typically $10^{-7} - 10^{-3}$ mm) but repeated cycles of loading may extend the crack to a stage where fracture occurs. The problem is very important industrially, because failures of large structures like trains, automobiles, oil drilling rigs and mining equipment can cost millions of dollars, and cause death and injury to personnel.

The computational model that we used is described in [7]. In this approach, finite element techniques are used to compute the stress field throughout the component for a given applied load in the absence of cracks. Then cracks are assumed to occur at critical boundaries of the component and a recent modification [8] of the finite element alternating method [9] is applied to compute the stress intensity factor at the crack positions. Finally the Paris growth law [10] is used to compute the number of loading cycles required for the cracks to grow from its given initial size to a given final size. The number of cycles at the worst (least cycles) crack is taken as the fatigue life. Because a fine numerical mesh is required at the crack tips, computation times are typically of the order of minutes even for small two-dimensional models.

For the purposes of the stress analysis and subsequent fracture mechanics the shape profile is defined by control points. The finite element analysis proceeds by fitting a spline through the points to generate further boundary points. For optimization purposes the control points may be considered the state variables in the search space. For example if a curve in two-dimensions is specified by 50 movable points then the optimization has 100 independent variables. Optimization in a space of such high dimension is notoriously difficult. A further difficulty is how to impose desired constraints; especially difficult are constraints on the slope of the boundary.

Despite these difficulties some success has been achieved using a "biological algorithm" which moves each control point normal to the boundary a distance based on the assessment of that point. Earlier work assessed points on the basis of stress [11] or stress intensity factor [12],[13],[14]; more recently [7] fatigue life has been used.

An alternative approach, which is proposed in this paper, is to use a family of curves or surfaces specified by some parameters and to generate the control points from the curve. Finding the optimum shape involves determining the best parameters. Since the number of parameters is small, optimization is more efficient. However for this approach to find the true optimum the family of curves must include that optimum. Construction of a suitable family of curves can be a difficult design problem.

3 The Test Problem

The test problem is an aluminium alloy plate of thickness 1.5mm with dimensions 320mm by 600mm as shown in Figure 1. A remote stress field of 100MPa is applied in the vertical direction. The plate contains a hole of width 20mm which is 10mm from the left edge. As the geometry is symmetrical about the horizontal mid plane only the top half is modeled. This simple test problem is characteristic of a number of real designs found in practice. For example, the plate between the rear seats and the trunk of a passenger car, typically has a section removed to allow long items to project from the trunk into the cabin. In general, the hole needs to be as large as possible without compromising the mechanical stability of the plate, which is an important part of the structure of the car. In another setting, the same basic design can be found in the fuel tanks of high performance jet fighters. In this case a mechanical stiffener is placed in the fuel tank, which is in turn held in the wing, to increase the stability of the wing structure. However, a hole must be placed in the stiffener to allow fuel to move freely. Again, the hole needs to be large enough to allow fuel to move, but not too large that the effect of the stiffener is compromised.

The aim of the experiment is to determine the hole shape for optimal fatigue life in the presence of cracks distributed around the hole profile. The cracks have an initial length of 1mm and are deemed to have failed when they reach a length of 5mm.

4 Parameterization

Theory predicts that optimal hole profile must be free of sharp corners and symmetric about the horizontal mid plane. A suitable family of smooth curves, symmetrical about the point (p, q), are the "generalized ellipses",

$$\frac{|x-p|^t}{a^t} + \frac{|y-q|^t}{b^t} = 1 \tag{1}$$

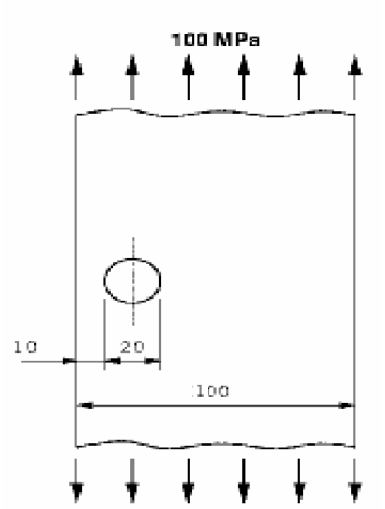


Fig. 1. Test problem for hole profile optimization

These are closed curves of width $2a$ height $2b$ where t controls the curvature at the shoulders of the curves. We will call this the "tightness". Figure 2a shows some of these with $a = 10$, $b = 20$ and a variety of values of t. More generally, the curves

$$\frac{|x-p|^t \exp(-t\beta(y-q))}{a^t} + \frac{|y-q|^t}{b^t} = 1 \tag{2}$$

also have width $2a$ and height $2b$ at the vertical axis. The inclusion of the exponential factor allows for asymmetry about that axis. The parameter β, called the "bias" here, controls the slope at the top. Figure 2b shows samples with $p = 0$, $q = 0$, $a = 10$, $b = 20$, $t = 5$ and various β.

For the plate model we employed $a = 10$, $p = 20$, and $q = 0$ giving a hole of width 20mm which is 10mm from the left boundary. Thus the variables b, t and β are the optimization parameters. The search space used was the domain $5 \leq b \leq 35$, $2 \leq t \leq 9$, $-0.02 \leq \beta \leq 0.02$.

5 Results

5.1 Experiment 1

Experiments were performed on a Compaq SC cluster, consisting of 20 Alpha EV68 processors connected by a Quadrics switch. Nine independent optimizations were performed starting from various initial parameters evenly spread through the search space. Each optimization used the simplex search algorithm [15]. The results are shown in Table 1.

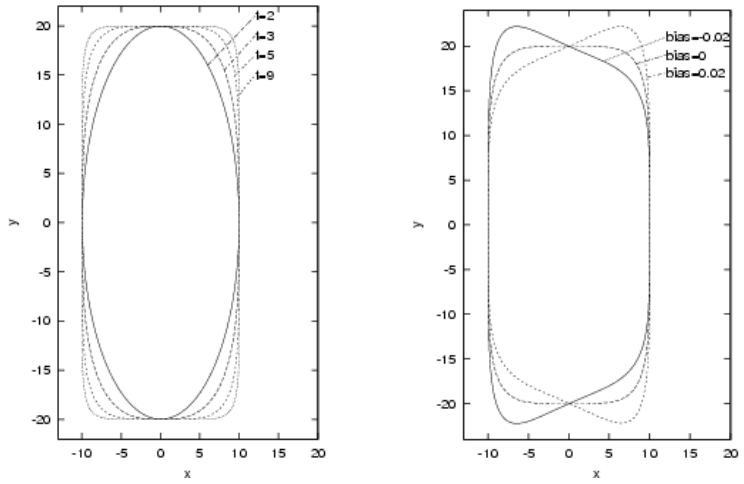


Fig. 2. Parametrized families (a) Equation (1), (b) Equation (2).

The "parameters" referred are the triples (b, t, β). As well as the final parameters and the optimum achieved, Nimrod/O reports statistics on the search process. The "Itrs" column gives the number of iterations of the simplex search. The "Evals" column contains two numbers; the second (in parentheses) is the number of evaluations required by the algorithm. However some evaluations may be duplicates, in which case the result is obtained from a cache rather than recomputed. The first number gives the number of unique evaluations. Since jobs are computed concurrently in batches, the number of batches gives a better indication of the running time of an optimization. The "Batches" column gives the number of batches requested by the algorithm and the number actually calculated. Because all nine optimizations were performed in parallel and batches averaged about 4 jobs, the concurrency was approximately 36 when compared with a serial optimizer performing the same searches.

Eight of the searches achieved an optimum with b at or close to the upper limit of 35. This agrees with previous work [13] on this problem using a control point algorithm that found the height of the hole grows to its upper bound. However one search (the first) revealed a local optimum in another part of the search space.

5.2 Experiment 2

So as to obtain a better understanding of the search space we used Nimrod to survey the search space evaluating the life at all combinations of $b = 5, 35, (5)$, $t = 2, 9, (0.5)$, $\beta = 0.020, 0.020, (0.002)$. Maximum stress on the hole profile was also computed in each case.

For each of these values of the bias β, rubber sheet representations were produced for both the life and the stress as functions of b and t. Further, isosurface

Starting Parameters	Optimal Parameters	Optim. Life	Itrs.	Evals.	Batches
(10,2,0)	(8.72,2.000,-0.00032)	3719	23	88 (104)	26 (26)
(10,5.5,0)	(35.00,4.390,-0.01687)	5335	19	76 (76)	19 (19)
(10,9,0)	(35.00,8.986,-0.00104)	5341	17	65 (68)	17 (17)
(20,2,0)	(33.86,4.078,-0.01904)	5361	18	70 (72)	18 (18)
(20,5.5,0)	(33.89,4.523,-0.01569)	5282	11	44 (44)	11 (11)
(20,9,0)	(35.00,7.963, 0.00148)	5316	12	48 (48)	12 (12)
(30,2,0)	(35.00,8.780, 0.00047)	5342	16	62 (64)	16 (16)
(30,5.5,0)	(35.00,8.300, 0.00000)	5342	27	31 (32)	8 (8)
(30,9,0)	(35.00,8.767, 0.00267)	5339	8	20 (20)	5 (5)

Table 1. Results of 9 simplex searches.

representations were produced for life and stress in terms of the three variable (b, t, β). These pictures are available at [16]. They show that the bias has only a minor effect on stress and life. For the case $\beta = 0$, the rubber sheets for life and stress are shown in Figures 3a and b respectively.

Figure 3a shows two regions of the search space that give high values for life. One is around $b = 35.0$, $t = 9.0$ (we call this region 1) and the other near $b = 5.0$, $t = 2.0$ (region 2). This explains why Nimrod/O searches found two distinct local optima. The fact that these optimal regions are on the search space boundaries suggest that extending the boundaries will allow better optima.

We expect that shapes with long life will have low stress. Figure 3b shows that this is true for region 1 but not for region 2. The latter case seems anomalous in that maximum life corresponds to maximum stress. This matter is resolved by visualizations of the stress field for shapes taken from these regions. Figure 4a shows the hole shape and associated stress field for the shape $b = 35.0$, $t = 4.0$ and $\beta = -0.020$, which gave the global optimum in region 1, and Figure 4b for shape $b = 5.0$, $t = 2.0$ and $\beta = -0.004$, the local optimum in region 2. The latter reveals that, although the shape gives high stress on the hole boundary, the stress falls off rapidly when moving into the body. Hence cracks on this boundary will initially grow quickly but this is balanced by very slow subsequent growth.

6 Conclusion

For the test geometry both long angular shapes and short rounded shapes produced high fatigue life, but only the former corresponds to low stress on the hole boundary. These results confirm that optimization for fatigue life may produce very different results from traditional optimization based on boundary stress. However, the current practice in industry is to only design for stress minimization. As a result of this work it would appear that to achieve optimal life it may be best to use a fracture/fatigue rather than a stress based optimization process. This has clear implications for both safety and economics of production.

For this problem the search space contained multiple local optima; this fact was revealed by performing multiple searches from various starting points. Nimrod/O facilitated this by performing these searches in parallel. To understand these results a full survey of the search space was required. Here Nimrod was an appropriate tool.

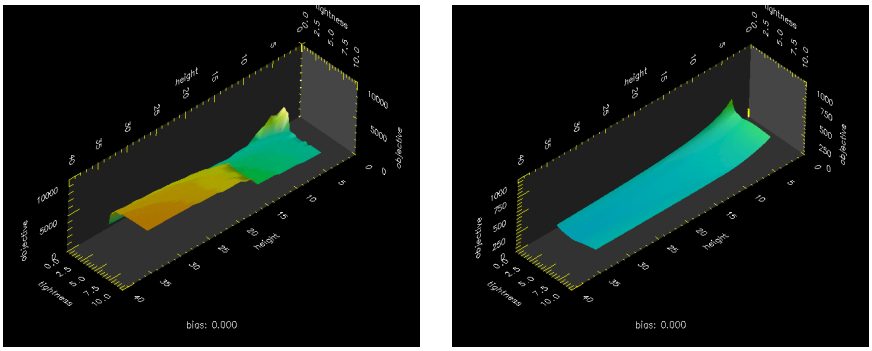


Fig. 3. Rubber sheet representations for (a) life (b) stress against b and t for $\beta = 0$.

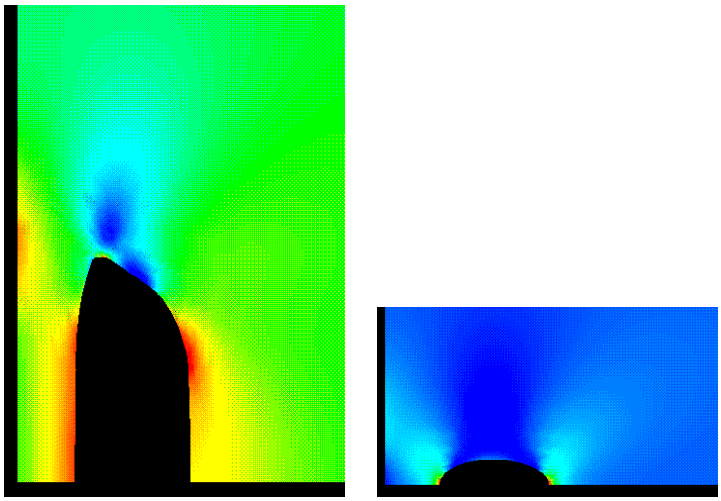


Fig. 4. Stress fields for hole shape (a) globally optimal (b) locally optimal.

References

1. Abramson D., Sosic R., Giddy J. and Hall B.: Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations. The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.
2. Buyya R., Abramson D., Giddy J.. Nimrod/G: an architecture for a resource Management and Scheduling system in a Global Computational Grid, 4th International Conference on High-Performance computing in the Asia-Pacific Region (HPC Asia), Beijing, May 2000, 283–289
3. Abramson D., Lewis, A. and Peachey, T., Nimrod/O: A Tool for Automatic Design Optimization, The 4th International Conference on Algorithms & Architectures for Parallel Processing (ICA3PP 2000), Hong Kong, December 2000.
4. Abramson D., Lewis, A. and Peachey, T. Case Studies in Automatic Design Optimisation using the P-BFGS Algorithm, 2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference, Seattle, Washington (USA), April 2001, 104–109.
5. Abramson D, Lewis A, Peachey T, Fletcher, C., An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics, SuperComputing 2001, Denver, Nov 2001.
6. Broek D., The practical use of fracture mechanics, Kluwer Academic Publishers, 1988.
7. Jones R., Peng D., Chaperon P., Tan M., Abramsos D. and Peachey T. Structural Optimization with Damage Tolerance Constraints, J. Theoretical and Applied Fracture Mechanics, (in press).
8. Jones R. and D. Peng, A simple method for computing the stress intensity factors for cracks at notches, Engng. Failure Analysis **9** (2002) 683–702.
9. Nishioka, T. and Atluri, S.N., Analytical Solution for Embedded Elliptical Cracks, and Finite Element Alternating Method for Elliptical Surface Cracks, Subject to Arbitrary Loadings, Engng. Fracture Mechanics, **17** (1983) 247–268.
10. Paris P. C. and Erdogan F. A critical analysis of crack propogation laws, Trans. ASME, J. Basic Engng., **85** (1963) 528–534.
11. Mattheck C. and S. Burkhardt, A new method of structural shape optimization based on biological growth, Int. J. Fatigue, **12** (1990) 185–190.
12. Chaperon P., Jones R., Heller M., Pitt S. and Rose F., A methodology for structural optimisation with damage tolerance constraints, Journal of Engng. Failure Analysis **7** (2000) 281–300.
13. Jones R., Chaperon P., and Sawyer J. P. G., Structural optimisation with damage tolerance constraints, Ageing Studies Lifetime Extension of Materials, , L. G. Mallinson (ed.), Kluwer Academic Press (2001) 601–608.
14. Jones R., Chaperon P. and Heller M., Structural optimisation with fracturestrength constraints, J. of Engng. Fracture Mechanics, **69** (2002) 45–67.
15. Nelder, J. A. and Mead R., A simplex method for function minimization, The Computer Journal, **7** (1965) 308–313.
16. http://www.csse.monash.edu.au/ donny/mc2/tcp/paper/

Case Studies in Automatic Design Optimisation using the P-BFGS Algorithm

David Abramson †, Andrew Lewis §, Tom Peachey †

† School of Computer Science and Software Engineering
Monash University,
Clayton, Vic 3168, Australia

§ High Performance Computing Facility
Griffith University,
Nathan, Qld, 4111, Australia

ABSTRACT

In this paper we consider a number of real world case studies using an automatic design optimisation system called Nimrod/O. The case studies include a photo-chemical pollution model, two different simulations of the strength of a mechanical part and the radio frequency properties of a ceramic bead. In each case the system is asked to minimise an objective function that results from the execution of a time consuming computational model. We compare the performance of an exhaustive search technique with a new non-linear gradient descent algorithm called P-BFGS. The exhaustive search results are produced using enFuzion, a commercial version of the parametric execution software Nimrod. P-BFGS is a parallel variant of the well-known BFGS algorithm and has been tested on a 64 processor Pentium cluster. The results show that P-BFGS can achieve a speedup when compared to the exhaustive search on 3 out of the 4 problems. In addition, it always uses fewer processors than an exhaustive search.

INTRODUCTION

Computational Science and Engineering (CS&E) involves the use of programs that find solutions to mathematical descriptions of real world problems. Where a simulation can be constructed that models the important physical properties of the real system it allows designers to explore a range of scenarios without the need to build a physical prototype. CS&E has been used extensively in the aeronautical and automotive industries, electronic and CAD and environmental modelling. CS&E is often computationally expensive, because, in order to achieve accurate modelling of a problem, it is important to use a high resolution in the mathematical decomposition. Thus, it is not surprising that supercomputers have been used extensively to facilitate CS&E. An important mode of use for CS&E models is in exploring some design space. When used in this way, it is necessary to run the same model many times, each time varying some key parameter. Distributed computers form an excellent platform for this type of work because the model runs concurrently on different data sets. An example of this mode of use is described in [6].

For some years we have been developing a software tool called Nimrod, which allows a user to consider a range of different input scenarios for an arbitrary computational model [1]. Nimrod uses a declarative "plan" file to describe the parameters that are to be varied and the tasks that are required to run the model. Parameters can be ranges or lists of simple types, such as integers, floats and strings. Nimrod takes the cross product of the parameter values and generates a unique set for each simulation run. It then runs the model on distributed computers in parallel. A "Grid Aware" version of Nimrod called Nimrod/G [4] can execute models on a distributed computational grid [2]. A commercial version of Nimrod, called EnFuzion, is targeted at machine clusters [23], and a spreadsheet interface for Nimrod is being developed, called ActiveSheets [7]. Nimrod and EnFuzion have been very successfully applied to a range of case studies in which CS&E models are run against a range of input scenarios, and the outcomes are compared and evaluated [14][6][3][17][4].

However, when there are a large number of different parameter values, the design space can become enormous. To combat this, we have been developing a new tool called Nimrod/O, which searches the space selectively rather than exhaustively with Nimrod [5]. Nimrod/O employs a number of different search heuristics, and one of these, P-BFGS, will be outlined in this paper. Nimrod/O attempts to minimise an objective function value, which is computed either directly by the CS&E model, or as a result of post processing the model output. This means that it can be used to answer design questions such as "which set of parameters will minimise the output of my model?".

In this paper we will discuss the details of a number of different case studies that we have trialed on Nimrod/O, and will also give details of the P-BFGS search algorithm that we have developed.

CASE STUDIES

Smog

SMOG is a photo-chemical pollution model originally developed by McRae et al. [20]. It has been applied to a number of real world environmental modeling exercises, including some Australian cities [16]. SMOG computes the concentrations of various chemical species in a 3 dimensional airshed. A number of sensitivity experiments are typically performed, in which the levels of pre-cursor species are scaled to determine the effect on the resulting pollution. The results of this work are often portrayed in a 2 dimensional "dodge" plot diagram, as shown in Figure 1, which shows how key pollutants such as Ozone vary as the pre-cursor species are throttled. When used in this case study we posed the following question: "What pre-cursor scaling factors will minimise the production of Ozone in the airshed?" The answer to this question is non-trivial because the chemistry is non-linear, and thus simply reducing the pre-cursor levels does not guarantee a minimum objective function.

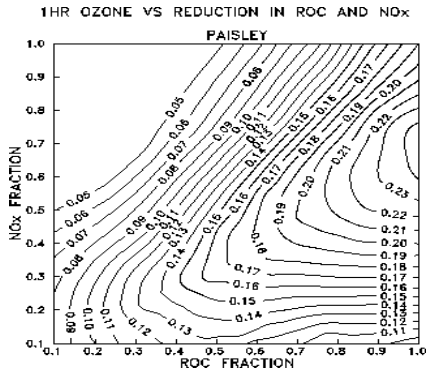


Figure 1 – A Dodge Plot of Ozone against precursors

Plate

"Damage tolerant" design is the design of load-bearing structures that contain flaws before being put into service. Software developed by Chaperon et al. [8] simulates such structures by performing finite element analysis with cracks placed at a range of locations. The PLATE model applies this software to a thin plate under biaxial loading containing a hole of a given width (Figure 2) to determine the hole shape that gives optimum static strength.

Previous work [8] used variable control points to specify the hole profile. The present study used instead a parametric model for the hole. Two variants were tried: S-Plate and GPlate. In the absence of cracks it is well-known an ellipse gives the optimal shape. S-Plate uses the family of ellipses with parameter a, $x = 10 \cos t$, $y = a \sin t$. C-plate allows more complex smooth shapes using three parameters $x = 10 \cos t$, $y = a_1 \sin t + a_2 \sin 3t + a_3 \sin 5t$.

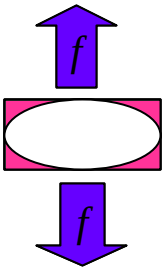


Figure 2 – Biaxially loaded plate with hole.

Ceramic

This case study concerns the design and characterisation of mobile telecommunication handsets. The structure of most mobile telephone antennas includes the chassis as an earth reference. Due to the chassis forming an integral part of the radiating structure, several difficulties are encountered in the measurement of mobile telephone antennas since they are generally connected to a network analyser via a coaxial cable. However, the coaxial cable used to feed a handset has significant effects on both the radiation pattern and resonant frequency [18].

A high permittivity ceramic bead on a straight wire transmission line can be used to suppress propagating cable currents at RF frequencies. This suggests the use of such a bead on the coaxial feed cable to minimise distortion of the radiation pattern during testing. In our antenna testing application, we wish to have minimum transmission through the bead at 1.0GHz. The FDTD method is used to perform a full-wave analysis of a cable structure. The dimensions and properties of the bead were optimised for minimum insertion loss to improve the effectiveness of the bead in minimising cable effects, and the structure is shown in Figure 3.

THE BFGS AND P-BFGS ALGORITHMS

BFGS

The general optimization problem is to minimize a real-valued function f of N variables. In the case of a numerical simulation, the N variables are the input parameters defining the search space to be investigated, and f is some "cost" function of the results of the simulation. In many cases it is necessary to reduce the raw results of the simulation from a large volume of data to a single, real value.

For the problems arising from engineering and scientific modelling, of the type discussed in Section 2, the values computed for f generally form a non-linear, continuous function, for which gradient descent algorithms can be applied. These methods use the derivative of the cost function, as well as its value, to select a search direction, essentially reducing the multivariate optimization problem to uni-variate minimisation along a search vector; a line search. Thus, they consist of two main operations that are executed repeatedly, gradient calculations and line searching. The algorithm chosen for this study is based on the quasi-Newton BFGS method, widely regarded as an efficient and robust gradient descent method for use with continuous functions [12][22][9][21].

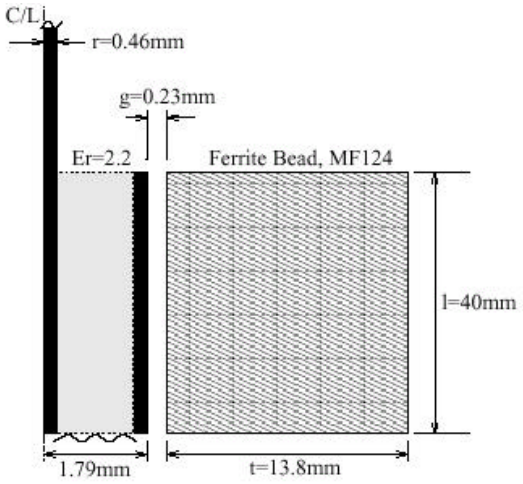


Figure 3 – ceramic bead simulation

The BFGS method maintains an approximation to the inverse of the Hessian, *H*, of *f*, where:

$$H_{ij} = \partial^2 f / \partial x_i \partial x_j$$

New approximations to the solution vector, *x*, are derived by:

1. Compute a search direction, $d = -H^{-1}\nabla f(x)$
2. Find a new *x*, using a line search, i.e. $x_+ = x + ld$
3. Update the inverse of H using the current approximation to *H*, *x* and x_+

For the practical implementation of the method used in the studies reported in this paper, convergence to the solution occurs when the cost function ceases to decrease below a user supplied tolerance, and when there is a sufficient decrease in the Euclidian norm of the gradient of the cost function from its value at the initial starting point. If the latter condition is not met, the method is restarted at least once with the Hessian reset to the unit matrix. The use of sufficient decrease in gradient norm to confirm convergence, in addition to small step change, is motivated by the idea that if the second derivative of the cost function is well conditioned at the terminating point, a small gradient norm implies a small error norm, and termination is reasonable.

P-BFGS – Parallel Optimization

Because most CS&E simulations are time consuming, we are interested in accelerating the search by parallelising the BFGS algorithm itself. The advantage of this approach is that no modification to the internal code of the simulations themselves is required; they can be treated as black boxes.

When the cost function is the computed result of a numerical simulation, finite difference approximations to the derivative are usually employed. The gradient calculation phase of the algorithm can be parallelised because the individual points in the finite difference stencil can be computed concurrently. The degree of parallelism achievable is determined by the dimensionality of the problem domain.

For unconstrained optimization the line search phase is often inherently sequential, but for the class of problems under consideration there will often be restrictions on the expected range of variables determined by physical constraints on their possible values. The restricted expected range of the variables allows the formulation of these problems as nonlinear optimization problems with simple bounds. The search vector is derived from the gradient and Hessian as in the standard BFGS method. A line segment is then constructed from the current point in the search space, in the direction of the search vector and extending as far as the nearest boundary of the search space. The line search phase of the algorithm may then be parallelised by using a method of interval subdivision, rather than proceeding with the sequential stepping methods typically used.

The parallel form of the algorithm can be summarized by the following stages of initialization, line minimization, convergence check and BFGS update. In pseudo-code these expand to:

Initialization

Evaluate the objective function at the starting point;

Calculate the gradient at the starting point by finite difference approximation (parallel function evaluation) and calculate the Euclidian norm of the gradient;

Set the inverse Hessian to the unit matrix;

Set the initial line direction to the inverse of the gradient;

Perform Line Minimization

While bracket width is greater than desired tolerance **do**

Truncate the line search vector for physical constraints;

Determine the nearest boundary in the search direction and set maximum excursion accordingly;

Sub-divide the interval as desired (usually an integer multiple of processors available);

Evaluate the objective function at the sub-intervals (parallel function evaluation);

Select a bracket of three points containing the minimum value;

end while

Test for Convergence

if (step change in function value and largest step change in position in any dimension are less than the desired tolerance)
then
Calculate the norm of the gradient at the "minimum"
if (norm of gradient subtracted from norm of initial gradient is less than an empirically determined constant)
then
optimization is complete **else**
if (this is not the first time convergence test at this point)
then
optimization is complete **else**
Reset the inverse Hessian to the unit matrix, reset the line search direction to the inverse of the gradient and repeat line minimization

Perform BFGS update

Calculate a new gradient at the line minimum

Calculate the step change in the gradient

Apply the BFGS update to the inverse Hessian

Calculate the new line search direction

Repeat Line Minimization and Test for Convergence

Inspection of the algorithm outline shows that the degree of parallelism is limited by the desired tolerance of the solution during the line minization phase, and to the number of dimensions during the gradient evaluation phase. During line minimization, interval sub-division is repeated until the sub-intervals are less than the desired tolerance. So for a given fractional tolerance, t, the maximum number of CPUs that can usefully be employed is $1/t$. Since most line searches will be less than the entire span of the search domain, to a first approximation the average number of CPUs usefully employed would be $1/(2t)$.

Unlike a sequentially stepping algorithm, the parallel algorithm is less prone to become trapped in some local minima. This is illustrated in Figure 4, which shows the sequence of points given by a sequential algorithm using a combination of stepping and parabolic interpolation, and by the parallel algorithm using interval sub-division. The sequential algorithm terminates in the first minimum found, while the parallel algorithm, extending its search to the boundary, finds the lower minimum.

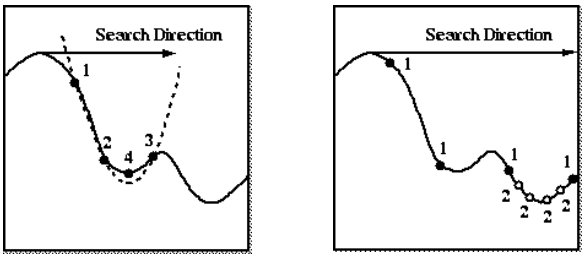


Figure 4: Sequential and parallel line minimization

Given the sources of concurrency in the algorithm, the speedup achieved through parallelisation of the algorithm alone is likely to be modest. However, because the surface is non-linear, it is necessary to perform more than one search in order to try and maximise the chance of finding the global minimum rather than just a local optimum. Accordingly, it is important to run the algorithm more than once, starting it at a different point in the search space each time. Because these searches are independent, they can also be performed in parallel, thus increasing the average speedup. This attribute will become evident in the next section, where we evaluate the performance of the P-BFGS algorithm on the case study problems.

PERFORMANCE RESULTS

Table 1 shows the results of performing an exhaustive search on each of the problems using EnFuzion. Whilst the parameters are continuous, by using a discrete step it is possible to transform the problem into a discrete one. Accordingly, this allows us to perform an exhaustive search across all of the discrete points. The “Func Evals” column shows how many times the program was run to achieve the result shown in the “Optimal Cost” column. The time in seconds is shown for a single execution, and the time for a total run is shown, assuming that it is executed across 64 processors. Whilst the S-Plate problem is quite fast, the

other 3 problems take quite significant times to solve. In particular, "Smog" requires 64 processors for nearly 2 months to produce a solution to the required accuracy.

Problem	Num Dims	# Func Evals	Optimal Cost	Time for 1 Func. Eval. (Secs)	Num of CPUs	Wall Clock Time (Hrs:Mins: Secs)
Smog	2	10000	68	31680	64	1375: 0: 0
S-Plate	1	201	15.0	204	64	0: 10: 40
C-Plate	2	2541	14.7	326	64	3: 35: 35
Ceramic	3	12896	-39.8	3000	64	167: 55: 0

Table 1 – Exhaustive Search of Problems

Table 2 shows the results of running the problems using the P-BFGS algorithm. All but the "Smog" problem were run 8 times (concurrently so that the wall clock time was the same as for a single run), but the P-BFGS algorithm performed so well on the "Smog" problem that it was only necessary to run it once.

For each problem, we show the best cost, worst cost and mean. An "*" in the "Opt" column indicates that at least one of the simultaneous executions managed to find the optimal cost, as reported in Table 1. The "# Func Evals for Best" column shows how many times the model was run in the case that produced the optimal result, and the "#Iterations for Best" shows how many times the algorithm recomputed the gradient.

The "Ave # Processors" is computed by taking the average number of processors for an individual run and multiplying it by the number of concurrent starts. All of these are less than 64 in comparison with the exhaustive search that used the entire machine.

The ratio of "Wall Clock Exhaust" to "Wall Clock Best" compares the time taken to compute the solution exhaustively using EnFuzion against using the P-BFGS search algorithm in Nimrod/O. Notably, Nimrod/O is able to find the optimal solution to "Smog" and "Ceramic" much faster than EnFuzion, and it does this with fewer processors. In particular, "Smog" only requires 5.1 processors on average to solve the problem 17.4 times faster! C-Plate is solved no faster with Nimrod/O than EnFuzion, but it does use fewer processors, and S-Plate is actually slower with Nimrod/O. However, given that the run only takes 34 minutes, the difference is not significant. "Smog" and "Ceramic" seem to perform well because the surfaces are quite smooth, and thus the gradient is a good indicator of the direction of the minimum.

CONCLUSION

In this paper we have attempted to do three things. The first is to introduce the area of using an automatic tool for performing design optimisation. Whilst this technique is not common, we believe it will become more popular as robust, flexible, CS&E models are developed. Second, we have described a set of real world case studies and how they can be addressed using a tool like Nimrod/O. Finally, we have presented a new parallel variant of the BFGS algorithm and showed how it can be applied to the case studies.

The results are encouraging in that they demonstrate that speedup is possible on some problems. However, it is likely that BFGS is not always the best algorithm to apply. For example, if the objective function is not smooth, then the gradient operator will not produce reliable search directions. Accordingly, we have built Nimrod/O in such a way that more optimisation algorithms can be integrated. This will be reported in a future paper.

ACKNOWLEDGEMENTS

The Nimrod/O project has been supported by the Australian Research Council, as part of an ARC Collaborative Grant. We wish to acknowledge our partners in this grant, namely the Environment Protection Authorities of Queensland, Victoria and Western Australia. We wish to thank Turbo Linux Inc. for access to the enFuzion software, and the Distributed Systems Technology CRC, for access to Nimrod/G. We also acknowledge our research partners in the Globus group, in particular Ian Foster from Argonne National Laboratories.

The case studies discussed in this paper involved the work of a number of colleagues. We wish to acknowledge:

- Dr Martin Cope, Dr Steven Spencer and Dr Maciej Skierski for their assistance with the Smog study;
- Professor Rhys Jones, Dr John Sawyer and Mr Paul Chaperon for their assistance with the Plate study;
- Mr Seppo Saario and Dr Jun Lu for their assistance with the Ceramic studies.

REFERENCES

[1] Abramson D., Sosic R., Giddy J. and Hall B., "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[2] Abramson, D. "Parametric Modelling: Killer Apps for Linux Clusters", Linux Journal, May 2000.

[3] Abramson, D., Foster, I., Giddy, J., Lewis, A., Sosic, R. Sutherst, R. and White. N., "The Nimrod

Computational Workbench: A Case Study in Desktop Metacomputing", Australian Computer Science Conference (ACSC 97), Macquarie University, Sydney, Feb 1997, pp 17 - 26.

[4] Abramson, D., Giddy, J. and Kotler, L. "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", International Parallel and Distributed Processing Symposium (IPDPS), pp 520- 528, Cancun, Mexico, May 2000.

[5] Abramson, D, Lewis, A. and Peachy, T., "Nimrod/O: A Tool for Automatic Design Optimization", The 4th International Conference on Algorithms & Architectures for Parallel Processing (ICA3PP 2000), Hong Kong, 11 - 13 December 2000..

[6] Abramson, D., Power, K., and Sosic, R., "Simulating Computer Networks using Clusters of PCs", HPC-TelePar'99 at the 1999 Advanced Simulation Technologies Conference (ASTC'99), April 11-15, San Diego, California,USA

[7] Abramson, D., Roe, P., Kotler L and Mather, D., "ActiveSheets: Super-Computing with Spreadsheets". 2001 High Performance Computing Symposium (HPC'01), part of the Advanced Simulation Technologies Conference, April 22-26, Seattle, Washington (USA).

[8] Chaperon, P, Jones, R, Heller, M., Pitt, S. and Rose, F. "A methodology for structural optimisation with damage tolerance constraints", Journal of Engineering Failure Analysis, 7, pp 281-300, 2000.

[9] Fletcher. R. "Practical Methods of Optimization." 2nd ed., Wiley, New York, 1987.

[10] Foster, I. and Kesselman, C. Globus: A Toolkit-Based Grid Architecture. In The Grid: Blueprint for a Future Computing Infrastructure, pages 259-278. Morgan Kaufmann Publishers, 1999.

[11] Foster, I., Kesselman, C., *Globus: A Metacomputing Infrastructure Toolkit*, International Journal of Supercomputer Applications, 11(2):115-128, 1997.

[12] Gill, P.E., Murray, W. and Wright. M, "Practical Optimization". Academic Press, London, 1981.

[13] Grimshaw, A. and Wulf, W., et al, "The Legion vision of a worldwide virtual computer", Communication of the ACM, 40(1):39-45, 1997.

[14] http://hathor.csse.monash.edu.au/RJK/

[15] http://www.globus.org/

[16] Lewis, A, Abramson, D and Simpson, R., "Parallel non-linear optimization : Towards the design of a decision support system for air quality management", IEEE Supercomputing 97, San Jose, 1997.

[17] Lewis, A., Abramson D., Sosic R., Giddy J., "Tool-based Parameterisation : An Application Perspective", Computational Techniques and Applications Conference, Melbourne, July 1995.

[18] Lewis, A., Saario, S., Abramson, D. and Lu, J., "An Application of Optimisation for Passive RF Component Design", Conference on Electromagnetic Field Computation, Milwaukee, June 4-7th, 2000.

[19] Litzkow, M., Livny, M. and Mutka, M.W., "Condor – A Hunter of Idle Workstations", proceedings of the 8th International Conference of Distributed Computing Systems, pp 104-111, June 1988.

[20] McRae G.J., Russell A.G. and Harley R.A., 1992. CIT photochemical airshed model -users manuals, Carnegie Mellon University, Pittsburgh, PA and California Institute of Technology, Pasadena, CA.

[21] Nelder, J.A., and Mead, R.,"A simplex method for function minimization", Comput. J., 7, pp. 308-313, 1965.

[22] Press. W. H., "Numerical Recipes: the Art of Scientific Computing". Cambridge University Press, 1986.

[23] Turbo Linux Inc, www.turbolinux.com

Problem	# Starts	Best Cost	Opt	Worst Cost	Mean Cost	# Iterations for Best	# Func Evals for Best	Wall Clock for Best (Hrs:Mins:Secs)	Wall Clock Exhaust / Wall Clock Best	Ave # of Processors required
Smog	1	68	*	-	-	2	46	79: 12: 0	17.4	5.1
S-Plate	8	15	*	15	15	2	46	0: 34: 0	0.3	36.8
C-Plate	8	14.7	*	15.7	15.1	7	240	3: 37: 11	1.0	48.0
Ceramic	8	-39.8	*	26.3	-2.1	5	61	15: 0: 0	11.2	27.1

Table 2 – Results of running P-BFGS on various case studies

Chapter 20
Parallel line search

T. C. Peachey, D. Abramson and A. Lewis

Abstract We consider the well-known line search algorithm that iteratively refines the search interval by subdivision and bracketing the optimum. In our applications, evaluations of the objective function typically require minutes or hours, so it becomes attractive to use more than the standard three steps in the subdivision, performing the evaluations in parallel. A statistical model for this scenario is presented giving the total execution time T in terms of the number of steps k and the probability distribution for the individual evaluation times. Both the model and extensive simulations show that the expected value of T does not fall monotonically with k, in fact more steps may significantly increase the execution time. We propose heuristics for speeding convergence by continuing to the next iteration before all evaluations are complete. Simulations are used to estimate the speedup achieved.

Key words: Line search, parallel computation

20.1 Line searches

A line search involves finding the minimal value of a real function f of a single real variable x. We attempt to locate the minimizing argument to within a "tolerance." Formally, given an interval $[a, b] \in \mathbb{R}$, a function $g : [a, b] \to \mathbb{R}$

T. C. Peachey
School of Computer Science and Software Engineering, Monash University, Clayton, VIC 3800, AUSTRALIA

D. Abramson
School of Computer Science and Software Engineering, Monash University, Clayton, VIC 3800, AUSTRALIA

A. Lewis
HPC Facility, Griffith University, Nathan, QLD 4111, AUSTRALIA

C. Pearce, E. Hunt (eds.), *Structure and Applications,* Springer Optimization and Its Applications 32, DOI 10.1007/978-0-387-98096-6_20,
© Springer Science+Business Media, LLC 2009

and a tolerance d, we require p, q such that $x^* \in [p,q] \subset [a,b]$ where g is minimal at x^* and $q - p \leq d$. We assume that the derivative, if it exists, is unknown.

Apart from their use in one-dimensional optimization, line searches are used in optimization on domains of higher dimension. For example, the quasi-Newton search methods use repeated cycles of determining the search direction and then performing a line search in that direction.

The line search algorithm is one of repeated subdivision of the interval and restriction to a subinterval. It can be summarized as follows:

1. Enter initial interval $[a,b]$ and tolerance d.
2. Set $p = a$, $q = b$.
3. Subdivide $[p,q]$ with points $p = x_0 < x_1 < x_2 < ... < x_k = q$, where $k \geq 3$.
4. Compute $g_i = g(x_i)$ for $i = 0, 1, 2, \ldots, k$.
5. Select $x_m : g_m = \min_i g_i$, the point where g is least.
6. If $m = 0$ replace p by x_0 and q by x_1,
   else if $m = k$ replace p by x_{k-1} and q by x_k,
   else replace p by x_{m-1} and q by x_{m+1}.
7. If $q - p \leq d$ then return (p,q),
   else go to Step 3.

Clearly the algorithm will terminate if $\sup(x_i - x_{i-1})/(q-p) < 1/2$, where the supremum is taken over both steps in the line search and iterations of that search. The process yields an interval $[p,q]$ which is guaranteed to contain the minimum if g is unimodal on $[a,b]$. Usually k is 3 as this is more efficient in terms of the number of function evaluations. It has long been known that the "Fibonacci search" [5] will minimize the number of function evaluations in the worst case. If g is approximately quadratic near the minimum then alternative methods such as Powell's [6] can be expected to be more efficient.

We are concerned with applications where each function evaluation may take at least several minutes on a fast processor. For example, g may represent aerodynamic drag on an object where x is some shape parameter, so a flow simulation would be required for each function evaluation. Further, we assume that batches of evaluations may be performed concurrently, on a cluster of computers or using the resources of the global grid. Clearly in such cases the speed of convergence may be improved by using more than three steps in each subdivision. These "parallel line searches" are the subject of this chapter.

20.2 Nimrod/O

Nimrod/O [1, 2] is an optimization package designed for the scenario described above, that is, long evaluation times employing multiple processors. The user prepares a "schedule file" such as the one in Figure 20.1. This specifies the problem parameters, any constraints linking them, how the objective

Fig. 20.1 A sample configuration file.

```
parameter alpha float range from 1 to 15
parameter tcmax float range from 0.5 to 1.5
parameter cmax float range from 0.5 to 1.0

constraint alpha >= tcmax + 2.0*cmax

task main
  copy * node:.
  node:substitute skeleton foil.inp
  node:execute run.all
  copy node:obj.dat output.$jobname
endtask

method simplex
  starts 5
    starting points random
    tolerance 0.01
  endstarts
endmethod
method bfgs
  starts 5
    starting points random
    tolerance 0.01
    line steps 8
  endstarts
endmethod
```

function is to be evaluated and the optimization algorithm to be used. This example uses two algorithms, the downhill simplex and the method of Broyden, Fletcher, Goldfarb and Shanno (BFGS), each run 5 times with different starting points.

The architecture of Nimrod/O is shown in Figure 20.2. Rectangles represent separate processes. The Controller reads the schedule and launches a process for each optimization. When an optimization requires a set of objective evaluations, it first checks the Cache to determine which jobs have already been run. Jobs that are new are sent to the dispatcher which is either the "Nimrod" system [1] or its commercial version "enFuzion." The dispatcher may run evaluations on the local machine, or on a cluster of machines or perhaps on the world grid.

Note that this architecture allows separate optimizations to be run in parallel. Within each optimization we have endeavored to speed the algorithms by employing parallel evaluations where possible. For example our implementation of the BFGS algorithm uses a parallel line search and also concurrent evaluations in the determination of the search direction; we call this implementation "Parallel-BFGS."

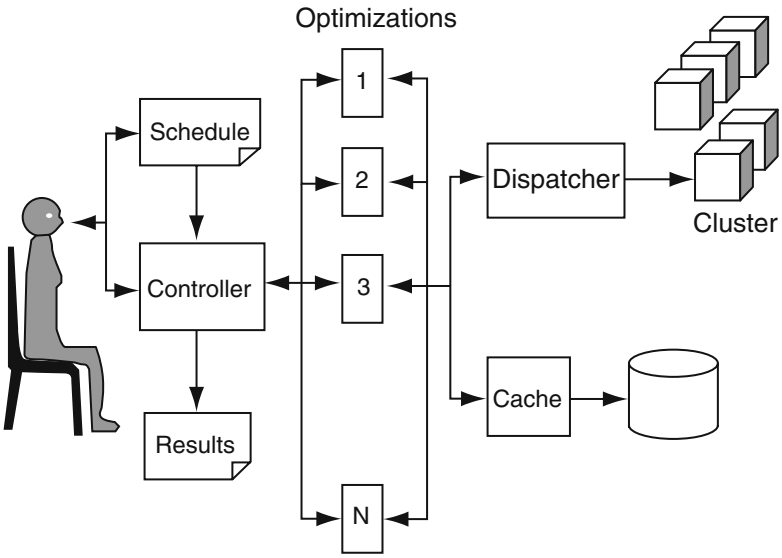


Fig. 20.2 Architecture of Nimrod/O.

Currently Nimrod/O is being applied in three areas:

- Design of an aerofoil. Here a two-dimensional aerofoil is specified in terms of three shape parameters. A FLUENT simulation is used to compute the flowfield around the aerofoil and compute lift and drag. The design problem is to determine the shape parameters that maximize the ratio of lift to drag.
- Optimal fatigue life. Finite element models are used to predict the life of mechanical components with pre-existing cracks under a cyclical stress regime. We require the component shape that maximizes this life
- Image compression. We consider a compression method based on the mammalian vision system which involves up to 96 parameters. The parameters are to be selected to minimize the compression ratio.

It was noticed during the aerofoil study that the execution time for evaluations was bimodal. Most jobs took about 30 minutes but occasional ones required between 3 and 4 hours. Consequently some of the line searches had completed all but one of the evaluations in less than 40 minutes and then required about 3 more hours to finish the last one. (There was no obvious pattern to the values of the domain that gave rise to long execution times.) This raised two issues:

A: A smaller number of steps in the line search may achieve faster convergence as fewer jobs are less likely to include an exceptionally long one.
B: Faster completion may be provided by a mechanism for aborting longer jobs and proceeding to a subinterval identified by the completed jobs.

We consider Hypothesis A in Section 20.3 and B in Section 20.4.

20.3 Execution time

20.3.1 A model for execution time

This section presents a model for the execution time for a line search, in terms of the number of steps used.

Suppose that each iteration of the line search uses $k \geq 3$ steps; we assume that the points are equally spaced. Let l be the length of the original search. Each iteration reduces the length of the current domain to a proportion $2/k$ of the previous (or $1/k$ if the minimum happens to fall at an end point). Let r iterations be the most required to reduce the length to the tolerance d so r is the least integer such that $l(2/k)^r \leq d$. Hence

$$r = \text{ceil}\left(\frac{\log(l/d)}{\log(k/2)}\right), \tag{20.1}$$

where ceil(x) signifies the least integer that is not less than x. We write T_i for the evaluation time for the ith subdivision point and assume that all the T_i have the same probability density function $f(t)$ and distribution function $F(t)$. We write s for the number of evaluations required in an iteration. Note that, after the first iteration, subsequent ones will not require evaluations at the end points of the subinterval. Further, if k is even and the best point in the previous interval was internal, then the objective at the midpoint of the current interval will have been found in the previous iteration. So we approximate s by $k-2$ if k is even and $k-1$ if k is odd. As these evaluations are performed in parallel; the evaluation time for one iteration is $B = \max_i T_i$. For the scenario discussed above these times are much larger than the times required for selection of the subdivision points and comparison of the values there. So we assume that the time for each iteration is just B. We assume also that the T_i are statistically independent. Under this condition, see for example [3], the distribution function for B is $F(t)^s$. Thus the mean time for completion of a batch is approximately

$$M = \int_0^\infty t \frac{d}{dt}\left[F(t)^s\right]\, dt. \tag{20.2}$$

Hence the expected time for the complete optimization is

$$E = Mr = \text{ceil}\left(\frac{log(l/d)}{log(k/2)}\right) \int_0^\infty t \frac{d}{dt}\left[F(t)^s\right]\, dt. \tag{20.3}$$

20.3.2 Evaluation time a Bernoulli variate

As a model of the bimodal distribution encountered with the wing flow experiments, consider the case where the execution time for a single job has a discrete distribution

$$f(t) = a\delta(t - x) + (1 - a)\delta(t - y), \tag{20.4}$$

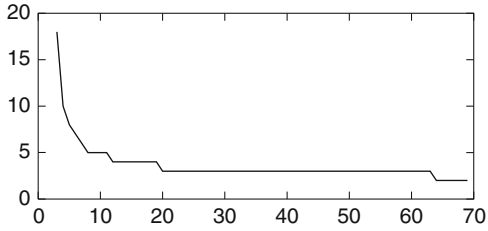
where δ is the Dirac delta and a, x and y are constants with $0 < a < 1$ and $x < y$. Then (20.2) becomes

$$M = xa^s + y(1 - a^s) \tag{20.5}$$

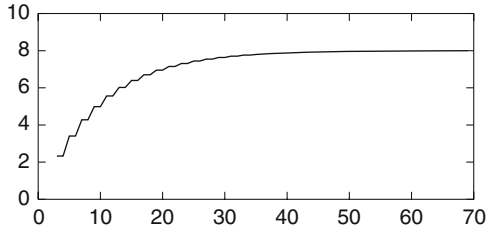
and (20.3) becomes

$$E = [xa^s + y(1 - a^s)] \text{ ceil} \left(\frac{\log(l/d)}{\log(k/2)} \right). \tag{20.6}$$

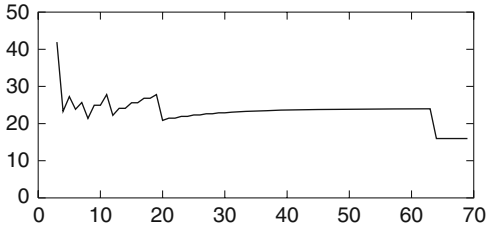
Graphs of these functions are shown in Figure 20.3. Figure 20.3(a) shows how r decreases in a piecewise manner. Figure 20.3(b) gives M for the case $x = 1$, $y = 8$, $l/d = 1000$ and $a = 0.9$. Figure 20.3(c) shows E, the product of r and M. Since M increases and r is piecewise constant, E increases while r is constant.



(a) Number of iterations, r



(b) Expected time per iteration, M



(c) Expected time for line search, E

Fig. 20.3 Performance with Bernoulli job times.

20.3.3 Simulations of evaluation time

For other distributions of job times, computation of (20.2) becomes difficult so we have performed simulations instead. The line search was performed on the function $g(x) = e^{-x}\sin(20x)$ on the domain $[0, 1]$, shown in Figure 20.4. This function has four local minima with a global minimum at $x \approx 0.2331$. The tolerance used was 0.001. Job times were generated randomly from (a) an exponential distribution with parameter 2 and (b) a rectangular distribution on $[0, 1]$.

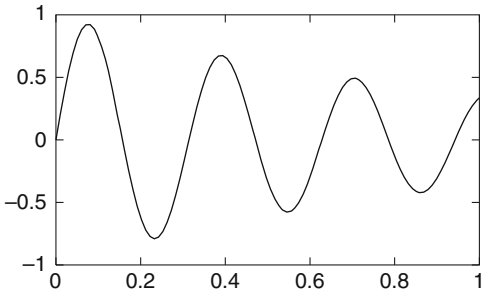


Fig. 20.4 Test function $g(x)$.

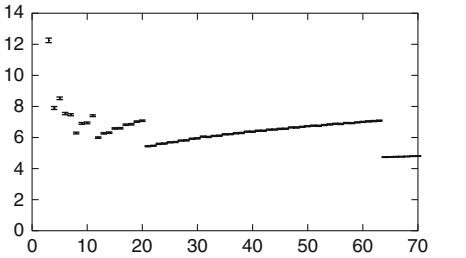
Figure 20.5(a) shows the mean total execution time, averaged over 10,000 runs, plotted against k for the exponential distribution. Each point is shown with error bars enclosing three standard errors. Figure 20.5(b) does the same for the rectangular distribution. Similar results were obtained for a wide variety of tolerance values.

For some simulations the line search failed to locate the global minimum, converging on a local minimum instead. Figure 20.5(c) shows the "effectiveness," the proportion of runs that achieved the global minimum. Here the algorithm is deterministic so effectiveness for a given k is either 0 or 1. In the next section the search will depend on the order of arrival of jobs and effectiveness will be fractional.

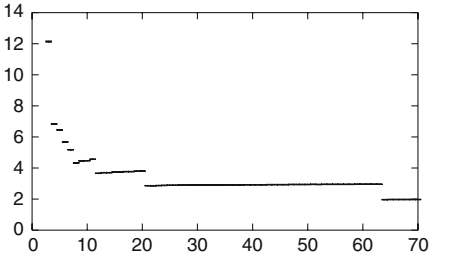
20.3.4 Conclusions

The preceding results show that increasing the number of steps in a parallel line search may be counter-productive; increases in k may produce considerable increases in E. For this to occur there must of course be variability in the job times. Note that Figure 20.5(b) shows much less increase than does Figure 20.5(a), although the mean and variance of the job times are similar. The significant factor is that the probability of job times is considerably larger than the mean.

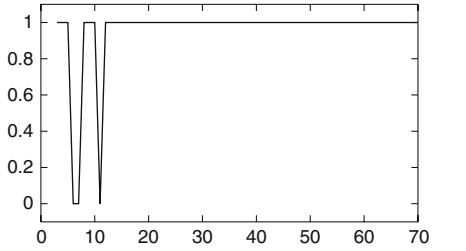
Fig. 20.5 Results of simulations.



(a) E versus k, exponential distribution job times



(b) E versus k, rectangular distribution job times



(c) Effectiveness versus k

A typical user of the line search algorithm will not have information on the distribution of job times. However the total time $E(k)$ has local minima at points where $r(k)$ decreases and these values can be predicted from knowledge of just the initial interval length l and the tolerance d. Consideration of (20.1) shows that r falls to a value ρ at $k = \text{ceil}\left(2\ \sqrt[\rho]{\frac{l}{d}}\right)$. This can be used to compute the number of steps k for a desired number of iterations ρ.

Our analysis has assumed that evaluation times are independent. If these are dependent, one may expect positive autocorrelation on the parameter space. This would lead to reduced variation in the later iterations of the line search which in turn would reduce growth in E between jumps. When the objective function is continuous but not unimodal we expect *a priori* that increasing the value of k makes attaining the global minimum more likely. Figure 20.5(c) supports this.

20.4 Accelerating convergence by incomplete iterations

20.4.1 Strategies for aborting jobs

We consider strategies for proceeding to the next iteration of a line search before the evaluations for all points in the current iteration are complete. Three heuristics are proposed.

Figure 20.6(a) illustrates a situation where 5 of the 7 evaluations of a function $g(x)$ are complete. The minimum so far occurs at $x = 3$ and the neighbors of that point have been evaluated. If g is unimodal then clearly the minimum is in the range $[2, 4]$. Thus the remaining evaluations may be aborted and the line search can proceed to the next stage. This leads to the following algorithm for one iteration of a line search:

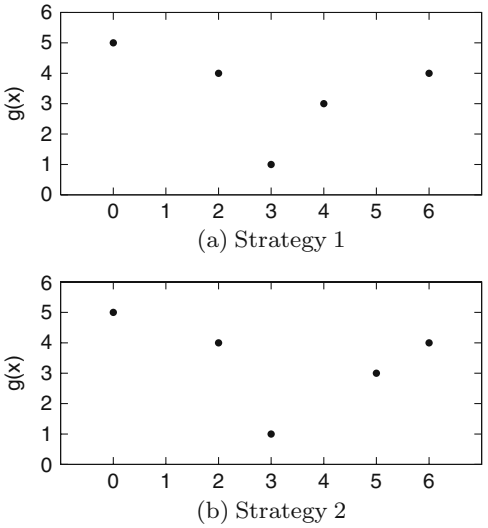


Fig. 20.6 Incomplete evaluation points.

Strategy 1.
Suppose an iteration involves determination of objective values $g_0, g_1, \ldots, g_k$. *At any time suppose that* S *represents the set of the* g_i *that have been completed by parallel evaluation. When each new value* g_j *arrives:*
add it to the set S
determine g_m, *the least value in* S
if $0 < m < k$ *and* $g_{m-1}, g_{m+1} \in S$ *return* $[x_{m-1}, x_{m+1}]$
else if $m = 0$ *and* $g_1 \in S$ *return* $[x_0, x_1]$
else if $m = k$ *and* $g_{k-1} \in S$ *return* $[x_{k-1}, x_k]$
continue

This approach can be extended to returning a greater interval than that provided by the immediate neighbors of the minimum point. In Figure 20.6(b), if g is unimodal then the minimum is in the interval $[2, 5]$; it may be worthwhile terminating the iteration with this interval. Many variants of this idea are possible. We investigate only the following.

Strategy 2.
Construct S as in Strategy 1. When each new value g_j arrives:
add it to the set S
determine g_m, the least value in S
if $0 < m < k$
if g_{m-1},$g_{m+1} \in S$ then return $[x_{m-1}, x_{m+1}]$
else if g_{m-1},$g_{m+2} \in S$ return $[x_{m-1}, x_{m+2}]$
else if g_{m-2},$g_{m+1} \in S$ return $[x_{m-2}, x_{m+1}]$
else if $m = 0$
if $g_1 \in S$ then return $[x_0, x_1]$
else if $g_2 \in S$ return $[x_0, x_2]$
else if $m = k$
if $g_{k-1} \in S$ then return $[x_{k-1}, x_k]$
else if $g_{k-2} \in S$ return $[x_{k-2}, x_k]$
continue

If sufficient processors are available it may be advantageous to both continue an iteration and to explore a subinterval identified as likely to contain the minimum. This leads to our third heuristic.

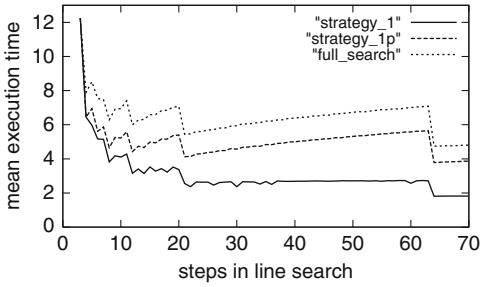
Strategy 3.
Use Strategy 2 to identify the subinterval and then start an iteration based on that interval, but also continue with the original iteration to completion. If later the original iteration finds a minimum better than any so far in the new iteration then the algorithm will "backtrack," abort the new iteration and start another iteration based on this improved minimum.

This is essentially a form of speculative computing, see [4]. Recursion allows a simple implementation.

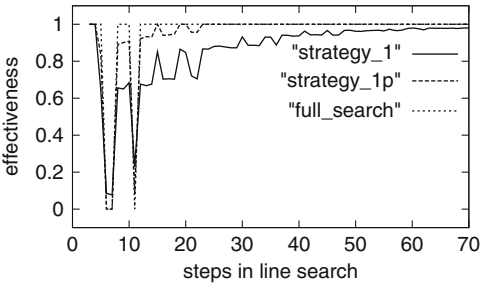
We also considered the effect of applying Strategies 1–3 only after the penultimate job has arrived, that is, when k of the $k+1$ evaluations have been completed. These heuristics will be denoted by 1p, 2p and 3p, respectively. A full search, completing each iteration before proceeding to the next, is denoted by F.

20.4.2 Experimental results

The strategies were implemented for line searches on the test function of Figure 20.4 with tolerance 0.001. For each k from 3 to 70, the search process



(a) Execution times



(b) Effectiveness

Fig. 20.7 Strategy 1 with exponential distribution of job times.

was simulated 10,000 times with execution times selected randomly from some probability distribution.

Strategies 1 and 1p were applied using exponential evaluation times with a mean $\lambda = 2$. Figure 20.7(a) shows the mean execution times and Figure 20.7(b) the effectiveness. In each case the results for these strategies are compared with those for a full search. Figure 20.8 shows times for the same range of strategies but with evaluation times from a rectangular distribution over the interval $[0, 1]$.

These experiments were repeated with the other strategies. Figure 20.9 shows results for the same method as Figure 20.7 but with Strategies 1 and 1p replaced by 2 and 2p. Similarly Figure 20.10 shows results for Strategies 3 and 3p.

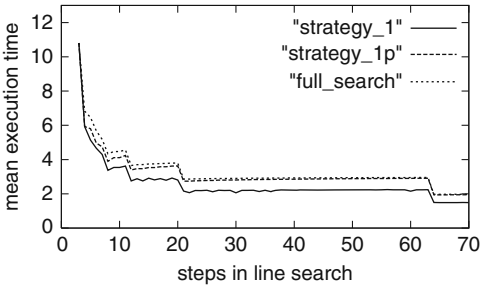
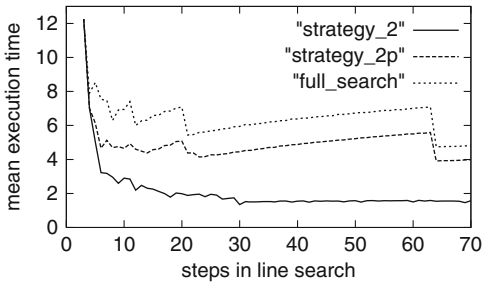
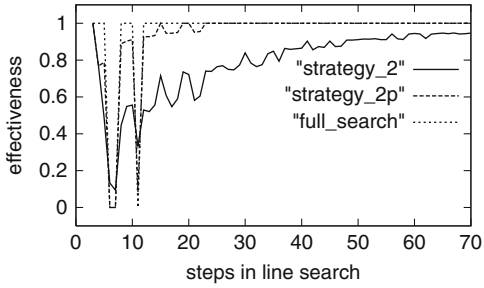


Fig. 20.8 Strategy 1 with rectangular distribution of job times.

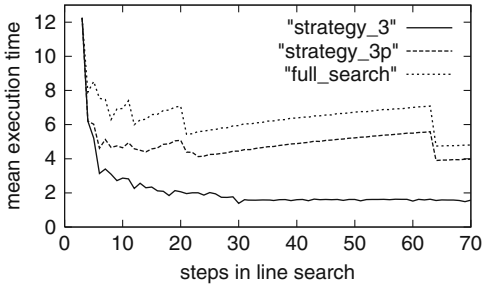


(a) Execution time

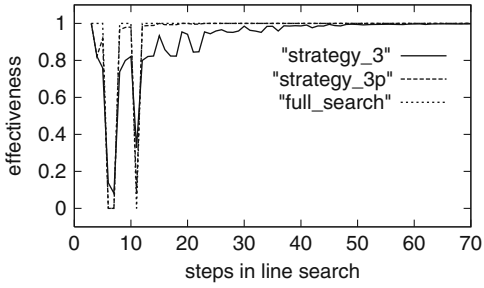


(b) Effectiveness

Fig. 20.9 Results for Strategy 2.



(a) Execution time



(b) Effectiveness

Fig. 20.10 Results for Strategy 3.

20.4.3 Conclusions

For job times with an exponential distribution, Strategy 1 shows a speedup of between Strategies 2 and 3 for $k \geq 12$, less for $k < 12$. This increased speed is at the expense of a deterioration in the effectiveness of the search. Strategy 1p is intermediate in performance between F and Strategy 1 for both execution times and effectiveness. The experiments with a rectangular distribution of job times showed less speedup, as there was less increase in M with k. Strategy 2 gave more speedup than that of Strategy 1 but with a further loss of effectiveness. Strategy 3 gave a speedup almost identical to that of Strategy 2 but with improved effectiveness. Hence this strategy is to be preferred when occasional long jobs are delaying execution. This advantage is at the expense of the need for extra processors when two iterations are running concurrently.

References

1. D. Abramson, I. Foster, J. Giddy, A. Lewis, R. Sosic, R. Sutherst and N. White, The Nimrod computational workbench: A case study in desktop metacomputing, in *Computational Techniques and Applications Conference*, Melbourne, July, 1995.
2. D. Abramson, A. Lewis and T. C. Peachey, An automatic design optimization tool and its application to computational fluid dynamics, in *Supercomputing 2001*, Denver, November, 2001.
3. A. O. Allen, *Probability, Statistics and Queueing Theory* (Academic Press, New York, 1978).
4. F. W. Burton, Speculative computation, parallelism and functional programming, *IEEE Trans. Comput.*, **C–34** (1985), 1190–1193.
5. J. Kiefer, Sequential minimax search for a maximum, *Proc. AMS* **4** (1953), 502–506.
6. M. J. D. Powell, An efficient method of finding the minimum of a function of several variables without calculating derivatives, *Comput. J.*, **7** (1964), 155–162.

Scientific Programming 14 (2006) 1–11
IOS Press

RSCS: A parallel simplex algorithm for the Nimrod/O optimization toolset

Andrew Lewis[a], David Abramson[b] and Tom Peachey[b]
[a]*Division of Information Services, Griffith University, Brisbane, Qld, Australia*
E-mail: a.lewis@griffith.edu.au
[b]*Department of Computer Science, and Software Engineering, Monash University, Melbourne, Vic., Australia*
E-mail: {davida,tcp}@csse.monash.edu.au

Abstract. This paper describes a method of parallelisation of the popular Nelder-Mead simplex optimization algorithms that can lead to enhanced performance on parallel and distributed computing resources. A reducing set of simplex vertices are used to derive search directions generally closely aligned with the local gradient. When tested on a range of problems drawn from real-world applications in science and engineering, this reducing set concurrent simplex (RSCS) variant of the Nelder-Mead algorithm compared favourably with the original algorithm, and also with the inherently parallel multidirectional search algorithm (MDS). All algorithms were implemented and tested in a general-purpose, grid-enabled optimization toolset.

Keywords: Parallel programming, optimization, Nelder-Mead algorithm

1. Introduction

In scientific and engineering research and design increasingly sophisticated, rigorous and realistic numerical simulations of physical systems are used to understand these systems, and aid in the design process. Engineers can use computational models instead of building physical prototypes to examine the behaviour of components or systems. Such an approach is usually faster and cheaper and hence allows the user to explore various design scenarios. In particular the user may search through combinations of design parameters in order to achieve a design that is optimal in some sense. Similarly, scientific research is increasingly using computational models, and there is often a need to determine those model parameters that produce the best fit to real-world data.

While use of computational models is becoming routine across a wider range of applications, they are often used in an informal manner: an engineer might use a simulation to test a handful of different cases and pick the best of them. More rigorous use to comprehensively explore the design parameter space has the potential to deliver better outcomes. This drives a demand for the capability to perform automatic optimization, minimising or maximising some derived quantity, a measure of "fitness" of the design, to reach a desired objective.

The computation of these objective function values is generally an extremely computationally intensive process when models, which may each take hours to compute, must be run tens, or maybe hundreds, of times to effectively refine possible designs. An ability to deploy the parallel and distributed computing resources that form the basis of contemporary high performance computing architectures would be a distinct advantage in making automatic optimization a practical tool in the engineering design process.

The design engineer who uses the optimization program, while an expert in the application domain, cannot always also be expected to be an expert in computer science. Ideally, to be useful a general purpose optimization tool should be easily applicable to a wide range of problems without assuming specialised knowledge in methods of optimization from the user. The more it can be treated as a "black box", the wider its potential adoption and the greater its end benefit. To meet these needs, the algorithms described in this word have

ISSN 1058-9244/06/$17.00 © 2006 – IOS Press and the authors. All rights reserved

been implemented as components of a fully integrated optimization toolset, Nimrod/O [1–3].

Many different optimization algorithms have been developed, from traditional gradient descent methods [4,5] to more recent innovations inspired by systems in nature, evolutionary and genetic algorithms [6]. Of enduring popularity, particularly for problems with "noisy" objective functions, or where gradient information is unreliable, unavailable or difficult to obtain, are direct search methods. Foremost among these is the simplex algorithm of Nelder and Mead [7].

In the Nelder-Mead simplex algorithm, the $n + 1$ vertices of a simplex of approximations to an optimal point in n-dimensional parameter space are sampled, ordered by objective function value, and an attempt made to replace the worst vertex by reflection through the convex hull of the remaining vertices, using limited sampling along the search direction so defined. Use of the Nelder-Mead simplex algorithm remains current, largely because, on a range of practical engineering problems, it is capable of returning a very good result [8]. It is also robust to small perturbations or inaccuracies in objective function values [9].

Since the original algorithm treats a single vertex at a time, the overall optimization process can be very time-consuming. With the wide availability of parallel and distributed computing resources, an obvious approach to attempt to reduce the total optimization time is to simultaneously consider and relocate several vertices. This paper proposes a method for concurrent execution of a simplex optimization algorithm, and presents results from numerical tests of the revised algorithm on a number of case studies derived from real-world problems in scientific and engineering applications.

2. The Nimrod/O toolset

Nimrod/O is a development of the Nimrod research project [10–12], incorporating automatic optimization into the framework of what was originally a parameter sweep toolset. Nimrod allows a scientist or engineer to succinctly describe their numerical simulation, define parameter ranges and perform automatic explorations of parameter space using enumeration of the cross-product of the defined parameters on parallel or distributed computers.

While an extremely useful tool, Nimrod suffers from the shortcoming of combinatorial explosion of required model evaluations as problem dimensionality and desired solution resolution increases. Nimrod/O avoids this problem by extending the toolset to include automatic optimization in the same, easily usable framework. Nimrod/O is equipped to run a number of very different optimisation algorithms, those described in this paper being just one set. Importantly, different problems are better suited to different algorithms, and Nimrod/O allows the user to try different algorithms (in parallel if the resources are available) and to choose the one that performs best. In this paper we are describing some enhancements to one of these algorithms.

The structure of Nimrod/O and outline of its operation is shown in Fig. 1. It is designed to be modular, the incorporation of additional optimization algorithms being a relatively simple operation. Since the C source code for Nimrod/O is freely available [13], the developer of a new algorithm can rapidly obtain the service of distributed evaluations Nimrod/O provides. Nimrod/O will also provide an extensive constraint parser, cacheing of completed jobs, granularity control for the parameters and statistical summaries of the evaluations.

The simplest way to incorporate an algorithm is to use the "hooks" provided. For example, if the algorithm is coded in the function `userSuppliedOptimization1` (for which a skeleton is provided) then that algorithm may be invoked by the line

```
method special 1
```

in a schedule file.

The user-supplied function will initially be passed a starting point in search space. What it does with this point is defined by the optimization method. For example, in simplicial methods the starting simplex is constructed around the starting point. However, it should be borne in mind that differing starting points are the only differentiation between the multiple, simultaneous optimizations available through Nimrod/O's multi-start feature.

The dimensionality of parameter space is available through a global variable, and its bounds through passed variable attribute structures. The code needs to use an array of structures of type `Point`. Each structure will hold the coordinates of a point in the search space; simultaneous evaluation of the objective function at a number of points constitutes a "batch of jobs". Execution of a batch is performed by passing the array to the function `evalBatchOfJobs`. On return each structure gives the status of the corresponding job, the result of any imposed constraints, the location of the "optimum" found and the value of the objective function at that point. More detail is supplied in Chapter 10 of the Nimrod/O User's Guide [14].

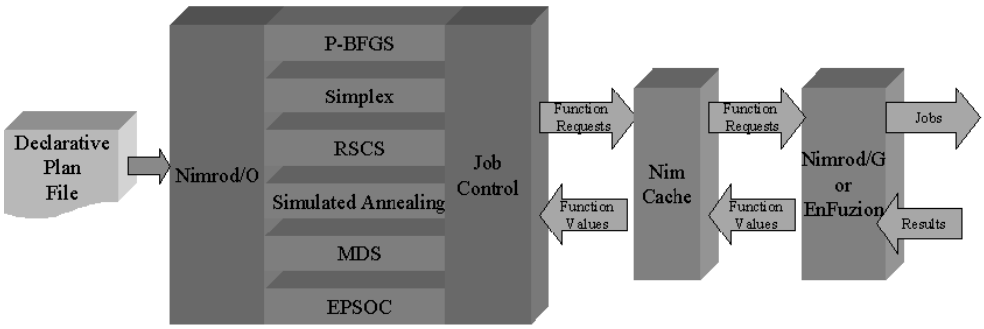


Fig. 1. The Nimrod/O Architecture.

A limitation of this approach is that only the tolerance setting, a value that can be used to decide convergence, is available to the algorithm. Other static, operational parameters of the optimization method may be read from a file, but if the developer wants settings to be passed from the schedule file to the algorithm, it is necessary to modify other parts of the Nimrod/O code. In this case it is advisable to fully incorporate the new method using an existing method as a guide.

Several changes are required. The files involved are shown in parentheses.

- Additions to the parser lexicon to recognise new statements in the schedule (parsesched.c and definitions.h).
- Additions to the structure `Opt_Settings` to include the information required (definitions.h).
- Insertion of parsing options in `ParseInnerLine` (parsesched.c).
- Additions of the new method as a case in `Optimize` (penalties.c).
- Creation of the new optimization algorithm to be called from `Optimize`.

A wide variety of methods is already available in the source code and can be used to provide examples of the code and information needed.

Control of optimization experiments is via a declarative plan file, a simple description of the execution of the numerical model and optimization problem, including parameters, their ranges, and details of the optimization methods to be used. Nimrod/O interprets the plan file, sets up the run environment and passes control to the requested optimization algorithm(s). Nimrod/O explicitly supports simultaneous execution of multiple optimization runs from different starting points, and simultaneous use of multiple optimization methods.

An optimization algorithm, as it runs, passes requests to Nimrod/O job control for objective function evaluations to be performed. These can be grouped in batches where the algorithm is capable of generating multiple concurrent tasks. Job control checks the requested set of parameters against a cache of previously computed results and then automatically dispatches evaluation jobs to computing resources provided by the user, whether they be parallel or distributed computers, via one of a number of job distribution mechanisms. Currently three are provided in the standard toolset:

- Nimrod/G, which is a Globus-enabled tool for use of computational grid resources [15]. Not only does the Grid offer additional computing resources, but potentially also access to specialised software packages and licences unavailable locally.
- an API to EnFuzion, a commercially available implementation of the original Nimrod tool that allows use of parallel computers or collections of workstations on a LAN.
- a mode in which processes are run locally using `fork`(). This mode suits symmetric multiprocessors.

These tools take care of job submission, execution tracking and input and output file handling.

Following is a simple, example declarative plan file. Further details of the syntax can be found in the Nimrod/O Users' Guide.

```
parameter b float range from 5 to 35
parameter tness float range from 2 to 9
parameter bias float range from -0.02 to 0.02

task main
   copy  runfiles/* node:.
   node:execute ./run.script $b $tness $bias
   copy node:result.dat  output.$jobname
endtask

method simplex
   starts 8 named "simplex"
```

```
      starting points random
      tolerance 0.005
  endstarts
endmethod
```

The plan file starts by defining the parameters. A parameter is named, its type defined, how it will be expressed and, where appropriate, the bounds on its value. Supported parameter types are float, integer and text. Float and integer types must be expressed as a range of values, text as a list of values. Where text parameters are used, the listed cases will be evaluated by simple enumeration, i.e. multiple optimizations will be performed, one for each value of the text parameter.

Then, in the section labelled as "task main", a brief description is given of how to run the numerical simulation. Input and output of the simulation are assumed to be via named files. This has been a common method of interaction with large-scale simulations, and is an easy method independent of the model implementation. For each objective function evaluation task input files are copied to the "node" on which execution is to be scheduled. All other files necessary for execution of the simulation should also be copied, including all necessary scripts and executables.

Then a user-provided script is run on the node. It is assumed that the script will provide the commands to run the simulation and derive the single, floating-point objective function value, possibly by post-processing of the model output. It may be noted that the parameters are provided to the script via named environment variables. These will be substituted by the particular parameter values supplied by the optimization algorithm at run-time. The results, written to a file, are copied back to the scheduling node, and placed in a file with the standard name "output" and an extension identifying the particular job.

Following the description of the model execution is the section specifying the algorithm to be used. Multiple algorithms can be specified, and will be executed simultaneously. Parameters for the algorithms are kept to a minimum, are generally intuitive, and provided with sensible default values. In the example shown, eight simultaneous runs will be performed using a parallel implementation of the Nelder-Mead Simplex algorithm, from random starting points. The algorithm description also specifies a desired solution tolerance. For the Simplex algorithm, and the new variant described in this paper, this specifies the convergence criterion by defining the magnitude of the fractional gradient of the final simplex.

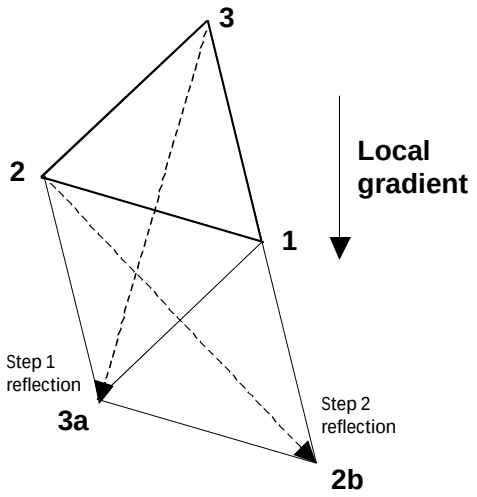


Fig. 2. Sequential Nelder-Mead reflection.

This is sufficient to specify the entire optimization experiment to Nimrod/O. Importantly, the user is not concerned with the computational platform, since support for this is provided by Nimrod or EnFuzion. Issues relating to the Grid are managed through the Nimrod Portal.

3. Reducing Set Concurrent Simplex (RSCS)

The application of supplementary search directions to the Nelder-Mead algorithm, drawing on the methods of the Multidimensional Search (MDS) algorithm of Dennis and Torczon [16–18] has been suggested previously by Hamma [19]. This still implemented additional searches sequentially. As illustrated in Figs 2 and 3, a straightforward, concurrent implementation of all the possible Nelder-Mead search directions is potentially inefficient.

When the search directions of the Nelder-Mead simplex algorithm are applied sequentially, step one producing the new vertex 3a and step 2 producing the new vertex 2b, the search directions are generally downhill, relative to the local gradient. However, if they are applied concurrently, it may quite often be the case that one or more may be to some degree uphill (for example, the trial vertex 2a in Fig. 3). Note: the MDS search directions have been omitted for clarity: they would be from vertices 2 and 3, through vertex 1.

In this paper a different approach to generating supplementary search directions for concurrent search is proposed. The search direction from the worst vertex is through the centroid of the remaining n vertices, as

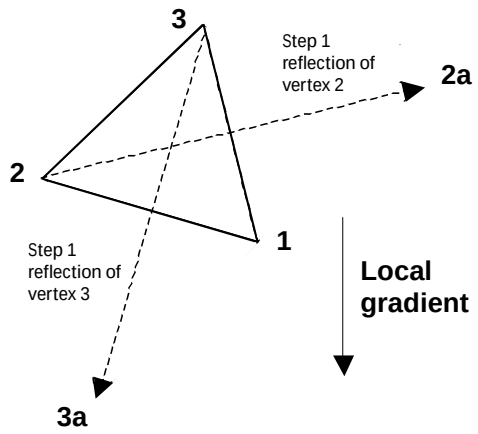


Fig. 3. Concurrent Nelder-Mead reflection.

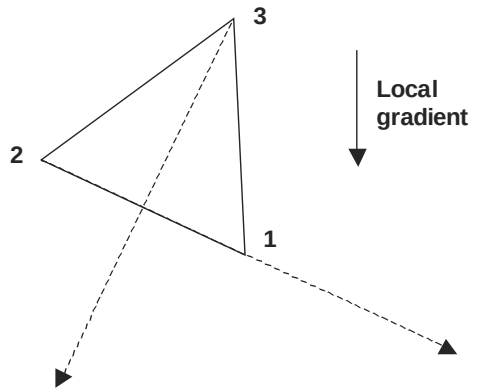


Fig. 4. RSCS search directions.

in the normal Nelder-Mead algorithm. But the search direction from the next worst vertex is through the $n-1$ remaining vertices that are better than it, and so on, until the search direction from the second best vertex is reduced to the MDS search direction though the best vertex. All searches are performed concurrently, and all vertices are independently relocatable.

The method can be considered as deriving from a hybrid of the Nelder-Mead and MDS algorithms. In this work it will be referred to as the Reducing Set Concurrent Simplex (RSCS) algorithm. The set of search directions generated is illustrated in Fig. 4. Comparing the search directions illustrated in Fig. 4 with the concurrent search directions of Fig. 3, it may be noted that the search directions generated by RSCS are more likely to be downhill.

Viewed in two dimensions RSCS can appear to be a simple hybrid of Nelder-Mead and MDS. However, if the 3-dimensional case is considered it becomes apparent there are differences. Figures 5, 6 and 7 show

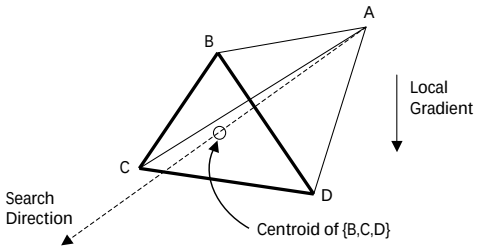


Fig. 5. First RSCS search direction – 3D case.

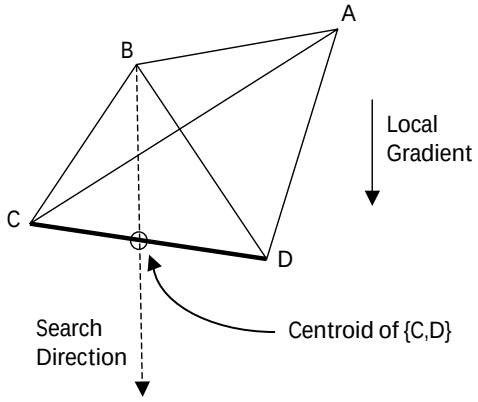


Fig. 6. Second RSCS search direction – 3D case.

the three search directions constructed by RSCS, and the edges and vertices used in their construction. The search direction shown in Fig. 6 will not appear in either the Nelder-Mead or MDS algorithms.

4. Numerical experiments

We have assembled a number of case studies drawn from interesting and challenging scientific and engineering applications. These were used to test and assess the performance of the individual algorithms. The following problems were used for these investigations:

4.1. Laser 1 and 2

A two-dimensional test surface was derived from the computation of a quantum electrodynamical simulation of a laser-atom interaction experiment [10]. The base case, Laser 1, is quite a smooth surface, the dataset containing only 4 minima, of which the global minimum is quite dominant, as can be seen in Fig. 8(a). Additive fractal noise was overlaid on this dataset to develop a “noisier”, more challenging surface to test

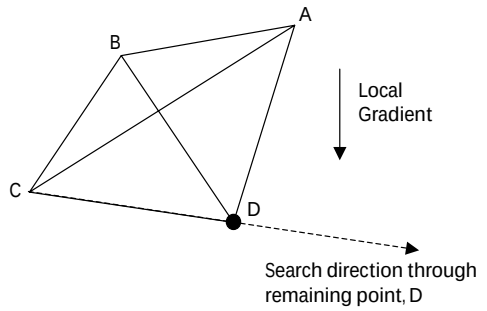


Fig. 7. Third RSCS search direction – 3D case.

the algorithms. This dataset, Laser 2, contained 1157 local minima of varying severity, and is illustrated in Fig. 8(b).

4.2. Crack 1 and 2

Finite element analysis of a thin plate under cyclic loading, with a cutout specified by parameters, was used to generate the Crack datasets [20]. Common practice in damage tolerant design has been to minimise the maximum stress under load. Isosurfaces of these stress values are shown in Fig. 8(c). This dataset, Crack 1, was reasonably smooth, with only 26 local minima. A new approach in modeling stressed components is to attempt to maximize durability. The Crack 2 model seeks to maximise the life of the part as determined by the minimum time taken for fatigue crack growth to a defined length from any of a number of starting crack locations. Isosurfaces at a number of values are shown in Fig. 8(d). In contrast to Crack 1, this dataset was "noisy", with 540 local maxima, and discontinuous isosurfaces.

4.3. Aerofoil

This test case models the aerodynamic properties of a two dimensional aerofoil. The objective function to be minimised is the lift-drag ratio [2], and this is computed by executing a Computational Fluid Dynamics model of the object. Figure 8(e) shows a number of isosurfaces in the parameter space investigated. The dataset was generally smooth, with only 12 local minima and a dominant global minimum.

4.4. Bead

The application from which this case study was drawn used a ceramic bead to minimise distortion of the radiation pattern of a mobile telecommunications handset during testing [21]. The objective function value, derived from an FDTD full-wave analysis of the handset and signal feed cable structure, was a measure of transmission strength through the bead at 1 GHz. The dataset for the Bead case study, of which isosurfaces for a particular value are shown in Fig. 8(f), is quite complex and contains 298 local minima.

4.5. Rosenbrock's function

In order to provide a point of comparison, the well-known Rosenbrock's function in two dimensions was included. The objective function values for this test case were directly computed from:
$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$ for $x_i \in [-2, 2]$ which has one local minimum at $f(1, 1) = 0$.

4.6. Case study assembly

It is generally not practical to directly use complex, real-world problems as test cases. The "black box", when queried, can take a considerable amount of time and computational resource to provide a response. So parameter sweeps were made of the numerical models that form the basis of these test cases, and the output data stored. These pre-computed data are interrogated, and linear interpolation employed to provide realistic responses from what are, in effect, "sandboxes" in which optimization programs can readily be tested. These "sandboxes" themselves represent a large investment of time and computational resource – for example, the data acquisition necessary to build the "Bead" test case required over 2 months of continuous computation on a multi-processor supercomputer.

Generally, the case studies fell into 2 sets:

- Smooth, with a dominant global minimum (Laser 1, Crack 1, Aerofoil, Rosenbrock's function)
- Multiple/many local minima, non-convex (Laser 2, Crack 2, Bead)

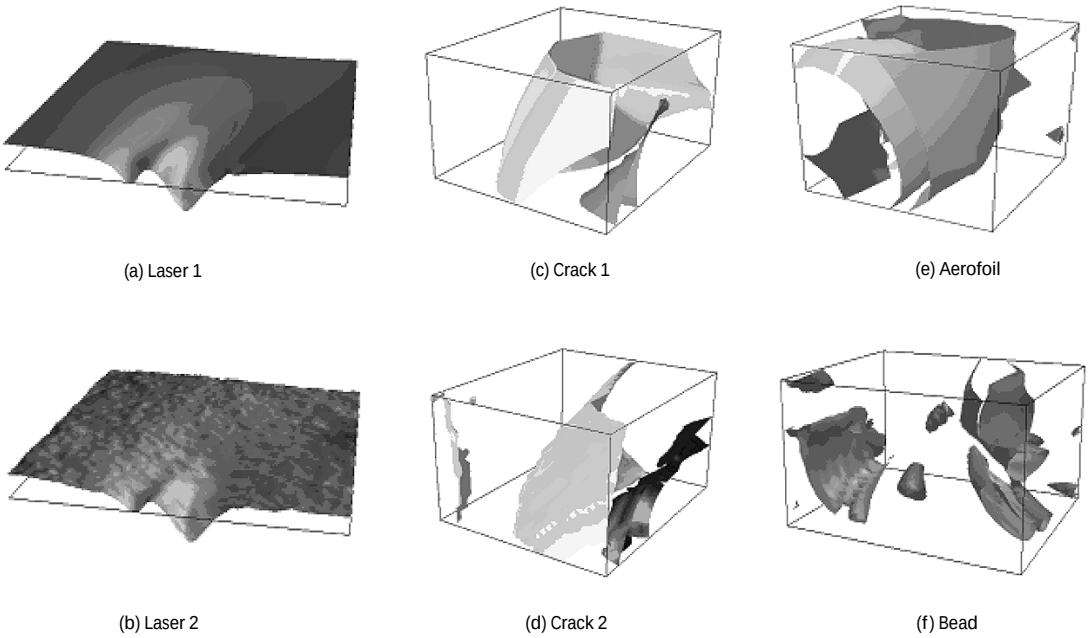
In formulating ideas about the most appropriate algorithm for use with a particular problem, a great deal may depend on whether the problem encountered is "noisy" or "smooth". Particular attention has been addressed to this issue when assessing algorithm performance.

Table 1
Median results obtained across 10 runs – Objective function values

	Laser 1	Laser 2	Crack 1	Crack 2	Aerofoil	Bead	Rosenbrock
NM	**−0.481**	−0.032	191.9	5299	−67.85	3.66	**0**
MDS	**−0.481**	0.279	**188.0**	**5319**	−67.92	**1.63**	0.066
RSCS	**−0.481**	**−0.286**	193.5	5310	**−68.56**	2.41	0.186

Table 2
Median results obtained across 10 runs – Function evaluations

	Laser 1	Laser 2	Crack 1	Crack 2	Aerofoil	Bead	Rosenbrock
NM	97	100.5	106	66	88	50	240
MDS	96	93	94	260.5	121	49	2358
RSCS	103	120.5	132	180	118	106	227



(a) Laser 1 (c) Crack 1 (e) Aerofoil

(b) Laser 2 (d) Crack 2 (f) Bead

Fig. 8. Test case isosurfaces.

5. Results of Experiments

To evaluate the new RSCS algorithm, it was compared with the original, Nelder-Mead algorithm and an implementation of the MDS algorithm. The Nelder-Mead algorithm performs evaluation of four points when considering relocation of each vertex, corresponding to the various alternatives of reflection, extension and contraction of the simplex. All these points are independent of each other, completely defined by the existing simplex geometry, and could be evaluated concurrently. The algorithm was slightly modified to exploit this concurrency, and all results reported for the "original" algorithm in this section refer to this parallel implementation.

Each of the algorithms was run on each of the test cases from 10 randomly distributed start points. For the purposes of comparison, in a given test case the same set of start points were used for each algorithm. The starting simplices were right simplices aligned with the coordinate axes. By default they were scaled to 10% of the parameter range for each coordinate, as use of reasonably large simplices has been shown to enhance performance [22]. Convergence criterion for most cases was a fractional step-wise gradient of 10^{-3}.

Function evaluations are performed concurrently in batches in Nimrod/O. The batch count can be interpreted as equivalent to Effective Serial Function Evaluations (ESFE), a measure of the wall-clock time taken for completion, providing the machine has enough pro-

Table 3
Median results obtained across 10 runs – Equivalent Serial Function Evaluations

	Laser 1	Laser 2	Crack 1	Crack 2	Aerofoil	Bead	Rosenbrock
NM	24.5	25.5	26.5	16.5	22	12.5	63.5
MDS	16.5	**16**	**11**	29.5	14	**6**	393.5
RSCS	**13.5**	**16**	12	**16**	**10.5**	10	**29**

Table 4
Best objective function values obtained in 10 runs

	Laser 1	Laser 2	Crack 1	Crack 2	Aerofoil	Bead	Rosenbrock
NM	**−0.48**	**−0.56**	**187.6**	5353	**−68.64**	−26.98	**0**
MDS	**−0.48**	**−0.56**	**187.6**	**5357**	**−68.64**	−16.12	3e-4
RSCS	**−0.48**	**−0.56**	**187.6**	5347	**−68.64**	−26.91	**0**

Table 5
Time taken, in ESFE, to achieve best objective function values, across 10 runs

	Laser 1	Laser 2	Crack 1	Crack 2	Aerofoil	Bead	Rosenbrock
NM	24	24	25	23	20	**16**	54
MDS	**16**	**12**	13	1000	14	7	1000
RSCS	22	**12**	**9**	42	**7**	12	**39**

cessors to concurrently evaluate all points, a reasonable assumption given the ready access to cheap clusters.

Analysis of the returned objective function values using the Shapiro-Wilk W test statistic determined that the results are not normally distributed. For this reason, median values and non-parametric, descriptive statistical methods are used for comparison of algorithms.

Tables 1, 2 and 3 show, for each algorithm on each test case over the 10 runs performed, respectively:

- The median objective function value obtained,
- The median number of function evaluations performed (FE) and
- The median Effective Serial Function Evaluations (ESFE).

For each test case, the **best median objective function value** is highlighted in **bold** type. Also highlighted is the **fastest time**, in terms of ESFE, for an algorithm to achieve a result within 10% of the best median objective function, as a percentage of the range of median values obtained.

Table 4 shows the best objective function value obtained in 10 runs for each algorithm on each test case. Table 5 shows the actual ESFEs required to obtain that result. For each test case, the best objective function value obtained by any algorithm, and the fastest ESFE to obtain that value, are highlighted in **bold** type.

From Table 1, it can be seen that RSCS appears to deliver slightly better results than Nelder-Mead on the majority of test cases. From Table 4 it can be seen that RSCS was also capable of equalling the best objective function value returned by both the Nelder-Mead and MDS algorithms on almost all test cases. In the one case it fell slightly behind, it was by less than 0.2%. Analysis of objective function values returned, using the Kruskal-Wallis H test statistic and pair-wise comparisons using Mann-Whitney U test statistics indicated there was no statistically significant difference in the quality of results returned by RSCS, Nelder-Mead simplex and MDS algorithms.

The only significant remaining difference between RSCS and the other algorithms is thus its speed. RSCS gave median ESFE that were consistently better than the Nelder-Mead algorithm. On average, RSCS was 78% faster.

From Table 3, it can be seen that RSCS is approximately 20% faster than MDS on average. This excludes the time taken by MDS on Rosenbrock's function, which was considered a pathological example of the tendency of simplicial methods toward premature convergence [23]. If the median time taken on Rosenbrock's function is included, RSCS is on average faster than MDS by a factor of 4.5. The decision to include these poorer results may be justified by reference to Table 4 in which it can be seen that on two different test cases MDS terminated by exceeding the maximum permissible iterations (1000), rather than satisfying convergence criteria. It can be conjectured that the insistence on congruency of consecutive simplices in the MDS algorithm forces premature contraction of the simplex diameter.

To further investigate the behaviour of the algorithms, the median objective function values achieved after each iteration of each algorithm across all 10 runs

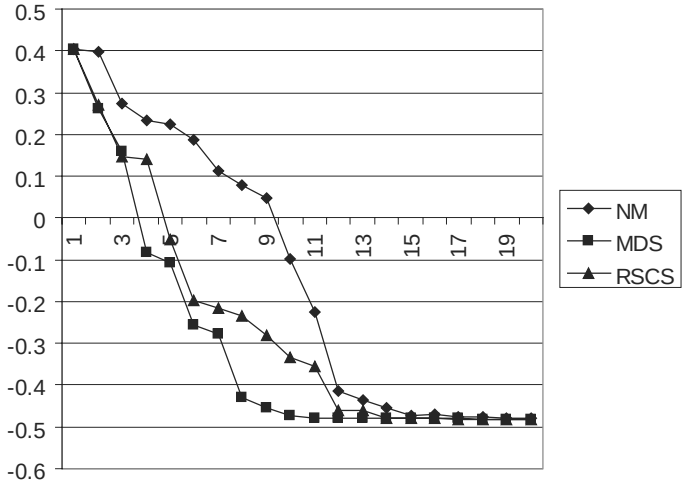


Fig. 9. Convergence history of median values – Laser 1.

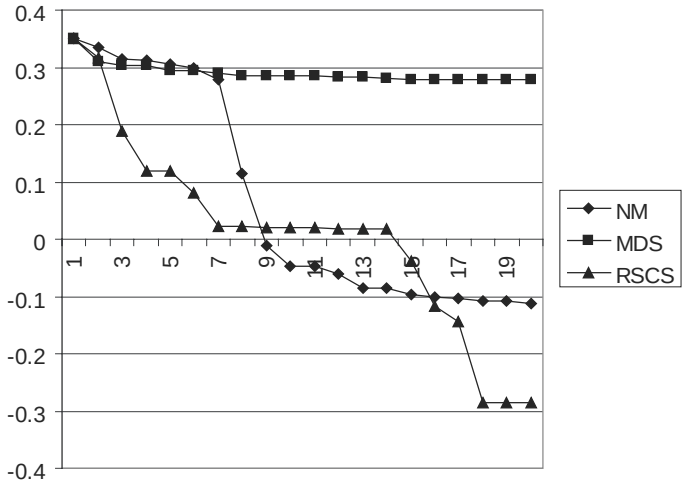


Fig. 10. Convergence history of median values – Laser 2.

on each of the real-world test cases were examined. As noted above, the case studies used can be classified into two classes, "smooth" and "noisy". In Figs 9 and 10 the two case studies derived from the quantum electrodynamical models, Laser 1 and Laser 2, are shown. The first of these exemplifies a "smooth" test case, the latter a "noisy" one.

From Fig. 9 it can be seen that all methods achieve similar results in terms of the objective function values obtained, as can be confirmed by reference to Table 1. Both RSCS and MDS reach better intermediate results faster than Nelder-Mead, and converge slightly earlier, demonstrating the advantages of treating multiple vertices simultaneously.

In Fig. 10, it is MDS that is left behind, as several of the test runs from which the median is drawn become trapped in local minima in regions of low overall gradient far from the global minimum. Nelder-Mead and RSCS obtain better results, with more of the RSCS runs terminating successfully at or near the global minimum. From these brief observations it may be concluded that RSCS is a more consistently reliable method, across

both smooth and noisy test cases.

6. Conclusions

A method of concurrent execution of a simplex optimization algorithm has been proposed, and its performance on a range of real-world problems compared with the popular Nelder-Mead and MDS algorithms from which it is derived. The supplementary search directions used are constructed from reducing sets of vertices in a manner which increases the probability they will be aligned with the local gradient.

In general, the Nelder-Mead and MDS algorithms perform well on the "smooth" test cases, but poorly on the class of cases with noise or many local minima. The new algorithm, RSCS, performs reasonably (or very) well on both types of problems. It generally provides equivalent or slightly better results, and delivers them considerably faster.

A general-purpose optimization toolset, Nimrod/O, in which the prototype and test algorithms have been implemented has also been described. The toolset makes the use of parallel and distributed computing for automatic optimization in the engineering design process readily and easily applicable. Nimrod/O can run a number of very different optimisation algorithms, in parallel if the resources are available. It allows a user to try different algorithms and select those which perform best on a particular problem. The toolset is configured for use on parallel computers, collections of workstations on networks, or grid-based computing resources.

By increasing the speed of the optimization algorithm using parallel computing resources, the method described delivers automatic optimization as a feasible tool for use by the practising engineer tackling real-world problems. With the growing availability of inexpensive computing clusters, rigorous exploration of design alternatives can be made, rather than haphazard, ad hoc evaluation of limited numbers of prototype designs.

References

[1] D. Abramson, A. Lewis and T. Peachey, *Nimrod/O: A tool for Automatic Design Optimization*, in Proceedings of The 4th International Conference on Algorithms & Architectures for Parallel Processing (ICA3PP 2000), Hong Kong, China, 2000.

[2] D. Abramson, A. Lewis and T. Peachey, *Case Studies in Automatic Design Optimisation Using the P-BFGS Algorithm*, in Proceedings of 2001 High Performance Computing Symposium (HPC'01), Seattle, WA, 2001, 104–109.

[3] D. Abramson, A. Lewis, T. Peachey and C. Fletcher, *An Automatic Design Optimization Tool and Its Application to Computational Fluid Dynamics*, in Proceedings of ACM/IEEE SC2001 Conference, Denver, CO, 2001.

[4] R. Fletcher, *Practical Methods of Optimization*, (2nd ed.), John Wiley & Sons, Chichester, 1987.

[5] P.E. Gill, W. Murray and M.H. Wright, *Practical Optimization*, Academic Press, London and New York, 1981.

[6] T. Bäck, *Evolutionary Algorithms in Theory and Practice*, Oxford University Press, New York, NY, 1996.

[7] J. Nelder and R. Mead, A simplex method for function minimization, *Comput. J.* **7** (1965), 308–313.

[8] M. Wright, Direct search methods: Once scorned, now respectable, in: *Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis*, D. Griffiths and G. Watson, eds, Harlow, UK: Addison Wesley Longman, 1995, pp. 191–208.

[9] H. Neddermeijer, G. van Oortmarssen, N. Piersma, R. Dekker and J. Habbema, Adaptive extensions of the Nelder and Mead simplex method for optimisation of stochastic simulation models, Faculty of Economics, Erasmus University, Rotterdam, The Netherlands, Tech. Rep., Econometric Institute Report EI2000-22/A, 2000.

[10] D. Abramson, R. Sosic, J. Giddy and B. Hall, *Nimrod: A Tool for Performing Parametised Simulations Using Distributed Workstations*, in Proceedings of the 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[11] A. Lewis, D. Abramson, R. Sosic and J. Giddy, *Tool-Based Parameterisation: An Application Perspective*, Proceedings of Computational Techniques and Applications Conference (CTAC95), Melbourne, Australia, 1995, 463–469.

[12] D. Abramson, I. Foster, J. Giddy, A. Lewis, R. Sosic, R. Sutherst and N. White, *The Nimrod Computational Workbench: A Case Study in Desktop Metacomputing*, Proceedings of the Australian Computer Science Conference (ACSC97), Sydney, Australia, 1997.

[13] D. Abramson, Nimrod/O, in Nimrod: Tools for Distributed Parametric Modelling, http://www.csse.monash.edu.au/~davida/nimrod/nimrodo.htm, Last viewed 7 November, 2005.

[14] T. Peachey, Nimrod/O user's guide: for version 2.1.x, Monash University, Melbourne, Australia, Tech. Report, 2003.

[15] I. Foster and C. Kesselman, Globus: A metacomputing infrastructure toolkit, *International Journal of Supercomputer Applications* **11** (1997), 115–128.

[16] J. Dennis and V. Torczon, Direct search methods on parallel machines, *SIAM J. Optim.* **1** (1991), 448–474.

[17] V. Torczon, *Multidirectional Search*, Ph.D. dissertation, Rice University, Houston, TX, 1989.

[18] V. Torczon, On the convergence of the multidirectional search algorithm, *SIAM J. Optim.* **1** (1991), 123–145.

[19] B. Hamma, Local and global behavior of moving polytope algorithms, CERFACS, Toulouse, France, Tech. Report TR/PA/97/39, 1997.

[20] T. Peachey, D. Abramson, A. Lewis and R. Jones, *Distributed Optimization Using Nimrod/O and Its Application to Fault Tolerant Structures*, Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM 2003), Czestochowa, Poland, 2003.

[21] A. Lewis, S. Saario, D. Abramson and J. Lu, *An Application of Optimisation for Passive RF Component Design*, Proceedings of the Conference on Electromagnetic Field Computation, Milwaukee, 2000.

[22] D. Humphrey and J. Wilson, A revised simplex search procedure for stochastic simulation response-surface optimization, in: *Proceedings of the 1998 Winter Simulation Conference*, D. Medeiros, ed., Piscataway, N.J.: IEEE Press, 1998, pp. 751–759.

[23] R. Barton and J.J.S. Ivey, Nelder-Mead simplex modifications for simplex optimization, *Management Science* **42** (1996), 954–973.

An Evolutionary Programming Algorithm for Multi-Objective Optimisation

Andrew Lewis
Griffith University
Nathan Campus
Brisbane, QLD 4111, Australia
a.lewis@griffith.edu.au

David Abramson
Monash University
Caulfield East
Melbourne, VIC 3145, Australia
David.Abramson@infotech.monash.edu.au

Abstract: This paper describes a new Evolutionary Programming optimisation algorithm and a method of its application to multi-objective optimisation problems. Computational results are presented demonstrating the algorithm's ability to find Pareto-optimal solutions for a real-world problem in radio-frequency component design.

1 Introduction.

In the engineering design process there is now widespread use of sophisticated and realistic numerical simulations of physical phenomena influencing design decisions. However, much of this use is on an *ad hoc* basis, manually investigating a limited set of design alternatives. Considerable improvements could be achieved with a capability to perform automatic optimisation, minimising or maximising some derived quantity, a measure of "fitness" of the design. This is an extremely computationally intensive process when models must be run tens, or maybe hundreds, of times to effectively search design parameter space. Current High Performance Computing systems mostly derive their capacity from parallel architectures so for optimisation methods to be practical and effective the algorithms used must preferably have a large degree of concurrency.

Over the past decade the computational capacity available to scientists and engineers has increased to the point where population-based methods of optimisation, in which many instances of a problem are treated simultaneously, have become practical for the solution of real-world problems (for surveys of methods and applications to multi-objective problems see Coello 1999 or Van Veldhuizen 1999). Moreover, in engineering design, the evaluation of the objective function is so much slower than the rest of the algorithm, that such codes are capable of giving excellent speedup in spite of the need for global communication on each iteration. **Genetic Algorithms** (GA) now may be frequently encountered in application to engineering problems (for examples, see Alander 1995.) Often the design needs of a particular engineering problem require the optimisation of more than one objective. In addition to describing a new Evolutionary Programming optimization algorithm, this paper outlines a method of applying the algorithm to such multi-objective problems.

2 Evolutionary computation

In the general case, evolutionary computation seeks to use insight into natural processes to inform population-based computational methods. These resulting Evolutionary Algorithms are popularly differentiated into three main classes (Bäck 1996) namely, Genetic Algorithms, Evolutionary Strategies and Evolutionary Programming.

In the general, population-based method, multiple instances of a problem, each represented by a vector of parameter values, are subject to various operators so that a population of problem instances in a "parent" generation evolve into a "child" population. This process is repeated through a number of generations.

Widely known, **Genetic Algorithms** are generally accepted as having been developed by Holland (Holland 1975). Genetic Algorithms, in contrast to Evolutionary Strategies and Evolutionary Programming, work on bitstrings of fixed length. For problems of continuous variable parameters the bitstring has a mapping to the vector of parameters, which generally implies they are capable of returning approximate, rather than exact, global minima. The bitstring representations are subject to processes of recombination through a crossover operator, an analogue of genetic inheritance in sexual reproduction, mutation and selection.

Evolutionary Strategies, a joint development of Bienert, Rechenberg and Schwefel in the 1960s (see, for example, Schwefel 1965), operate on continuous parameters, using normally distributed mutation and recombination. Recombination is either discrete, choosing which parent will contribute each of the parameter values in turn to a child, or intermediate with child parameter values being formed from the (weighted) mean of parent parameters.

Evolutionary Programming was developed by Fogel (Fogel 1962) and refers to that class of methods in evolutionary computation that apply a (uniform) random mutation to each member of a population, generating a single offspring. However, unlike other methods, no

recombination operators are applied. Population members may be considered as representative of species, rather than individuals, so phenotypic effects are emphasised instead of genetic change. After mutation, selection takes place, and half the combined population of parents and offspring enter the next generation. Such methods are generally simple, robust and highly parallel. Underused for many years, they were further developed in the 1980s and their use became more widespread in single-objective optimisation. However, they remain underused in multi-objective optimisation. A 1999 review of 272 publications in multi-objective evolutionary algorithms found only one citation related to Evolutionary Programming (Van Veldhuizen 1999).

3 Self-organised Criticality

The theory of self-organised criticality gives an insight into emergent complexity in nature (Bak 1996, Bak 1993). Systems in stable equilibrium exhibit linear behaviour. The system's response to a disturbance is proportional to the size of the disturbance. Large fluctuations can only occur if several factors simultaneously combine to act in the same direction, which is unlikely to occur. Such a system, then, is also unlikely to adapt rapidly to the demands of an objective function.

At the other end of the spectrum, chaotic systems can react violently to change, as small perturbations of initial values are amplified in the system's response. Chaotic systems have no memory of their past states and cannot evolve.

However, at the transition from a stable system to chaos, complex behaviour can emerge. It is this critical state that may deliver efficient adaptation. Self-organised critical systems evolve to the critical state without any external organising force. This is advantageous because it implies no *a priori* information about the internal functioning of the systems is required to develop an effective means of optimising an objective function expressed in terms of externally exposed parameters and observed system response.

Bak contended that the critical state was "the most efficient state that can actually be reached dynamically". Inspection of many natural phenomena suggests the critical state is capable of efficient adaptation to environmental pressures using simple, robust systems. If optimisation is considered as the adaptation of a system described by its parameters to the selective pressure of an objective function, then it appears developing a critical state may be a highly effective method of optimisation.

Bak sought to model evolution of species with a simple model of inter-species interactions and selection. A number of species were arranged, arbitrarily and randomly, in a ring topology. At each time step, the least fit species and its two neighbours in the ring were replaced by randomly instantiated new species. This model was demonstrated to lead to complex behaviour, with gradual evolution of the fitness of the whole population.

Self-organised criticality is also exhibited by the "sandpile model" (Bak 1996). In this model, "grains" of sand are modelled numerically, stacking upon each other and toppling onto neighbouring stacks under simple rules. The structure of this simple system can be observed to evolve to an organised state with dynamics characteristic of criticality.

4 A New Evolutionary Programming Algorithm (EPSOC)

In earlier applications of self-organised criticality to optimisation, it has been proposed that a separately computed power-law extinction rate be imposed on a spatial diffusion model, or cellular GA (Krink 2001)**.** Krink and Thomsen's model used pre-computed, stored dynamics of a sandpile model to control the size of extinction events in a diffusion model. The algorithm apparently does not attempt to evolve a population in a critical state, but indirectly imposes the observed behaviour of such a population.

In both the Bak-Sneppen nearest-neighbour, punctuated equilibrium model, and Krink and Thomsen's spatial diffusion model the population members are artificially arranged spatially: in a ring in the former, and in a toroidal, 2D grid in the latter. In contrast, by considering the trial solution parameter vectors as defining a location in an *n*-dimensional parameter space, the spatial behaviour of the Bak-Sneppen model is realized naturally in EPSOC.

In the following section we outline a new algorithm, called EPSOC, which is largely a straightforward implementation of Bak's model as an optimisation algorithm. It diverges in applying a high degree of greediness to the algorithm. Maintaining a large "elite" (in EPSOC, half the total population) can be viewed as a "constructive" operator. An analogous operator is "Maxwell's demon". Elitism has been clearly demonstrated to improved the performance of a GA (Zitzler 2000).

4.1 Single Objective Implementation

Restated, the general optimisation problem is:

$$\text{Minimize } f(\mathbf{x}) \text{ where: } f : \Re^n \rightarrow \Re^1 \text{ is an arbitrary non-linear function and } \mathbf{x} = \{x_0, ..., x_i, ..., x_n\}, x_i \in \Re \quad (1)$$

For a population-based method, the population, p, consists of a set of parameter vectors, $\mathbf{x}$: $p = \{\mathbf{x}_0, ..., \mathbf{x}_i, ..., \mathbf{x}_m\}$. For the real-world engineering design problems being considered, values for $f(\mathbf{x})$ are generally derived from execution of complex numerical simulations requiring considerable computation time.

The steps of the EPSOC algorithm are:

1. Initialise a random, uniformly-distributed population, p, and evaluate each trial solution, $\mathbf{x}_i \; \forall \; i$.
2. Sort the population by objective function value, $f(\mathbf{x})$.
3. Select a set, B, of the *nbad* worst members of the population. For each member of B, add to the set its two nearest neighbours in parameter space that are not already members of the set, *or from the best half of the sorted population.*
4. Apply a random, uniformly-distributed mutation to the selected set, B, i.e. re-initialise them. For all other members of the population, generate a "child" by applying a small (~10% of parameter range), random, uniformly-distributed mutation to the "parent" member.
5. Evaluate each new trial solution, $f(\mathbf{x})$.
6. If a child has a better objective function value than its parent, replace the parent with the child.
7. Repeat from step 2 until a preset number of iterations have been completed.

As each set of parameters defining a trial solution is independent of all others, it is immediately apparent that the evaluation of trial solutions at steps 1 and 5 can be performed concurrently. Since the evaluation of the objective function completely dominates the execution time, from Amdahl's Law we can expect extremely high parallel efficiency.

In a previous study we applied the single objective implementation of EPSOC and compared to a range of optimization algorithms (Lewis 2003), including a GA (Genesis 5.0, Grefenstette 1984), a parallel gradient descent method (P-BFGS), Dennis and Torczon's MDS, the Simplex method of Nelder and Mead, a Reducing Set Concurrent Simplex (RSCS), and line-searching variants of Simplex and RSCS.

All the algorithms were tested on six case studies drawn from real-world problems, from antenna and aerofoil design to quantum electrodynamical simulations of laser-atom interactions. Each algorithm was run multiple times on each test case. The GA and EPSOC were limited to a maximum number of iterations determined empirically to be equivalent to the time the other algorithms took to converge. On each test case, tests were performed to determine optimal operational parameter settings for the GA and EPSOC. It was noted that statistically significant differences in results returned for different parameter settings were more common for the GA than for EPSOC, i.e. EPSOC was less sensitive to parameter tuning.

Shapiro-Wilk testing of the distributions of returned values indicated the results were not normally distributed, so non-parametric, descriptive statistical methods were used to analyse the results.

The Kruskal-Wallis H test statistic was used to rank the algorithms on each test case, and determine if any algorithm performed significantly better than others. A pair-wise comparison was performed on the two highest-ranked algorithms on each case, using the Mann-Whitney U test statistic. In 4 of 6 test cases EPSOC was demonstrated to be the best algorithm *on these test cases drawn from real-world problems*, to a significance level of better than 0.05. In the remaining cases there was no statistically significant difference between the performance of EPSOC and the other leading algorithm.

4.2 Multi-Objective Optimisation

When tackling real-world problems, particularly in the field of engineering design, the desired optimal design may not be expressed in terms of a single objective. Product designers may wish to maximise some element of the performance of a product, while minimising the cost of its manufacture, for example. Different objectives may be conflicting, with little *a priori* knowledge as to how they interact. In the past, common practice was to optimise for a single objective while applying other objectives as penalty functions or constraints, a less than ideal approach. Evolutionary Algorithms (EA) have recently become more widely used for their ability to work well with large-scale, multi-objective problems (Van Veldhuizen 1999).

In general, there remains a role for a human Decision Maker (DM) in choosing between competing objectives. Multi-objective optimisation algorithms can broadly be categorised by when in the optimisation process the DM intervenes (Zitzler 1999, Van Veldhuizen 2000):

- **Decision making before search**: the DM aggregates the objectives into a single objective, including preference information (or weights). The problem is essentially reduced to a single objective optimisation.
- **Decision making during search**: the DM interactively supplies preference information to guide the search
- **Decision making after search**: the DM selects from a set of candidate solutions resulting from the search.

It may be noted that the first two of these approaches would seem to require some *a priori* knowledge of the problem domain in order to effectively provide preference information, particularly for methods involving aggregation. It has been asserted (Zitzler 1999, Coello 1998) that the first approach does not return Pareto-optimal solutions in the presence of non-convex search spaces. The multiobjective EPSOC method described in this paper (EPSOC-MO) falls into the last category.

EAs applied to multi-objective optimisation need to address two main problems:

- How is fitness assignment and selection performed?
- How is a diverse population maintained, to aid search space exploration?

Fitness assignment can conceptually be divided into a number of approaches (Zitzler 2002):

- Aggregation-based
- Criterion-based
- Pareto-based

A great deal of recent work has concentrated on the use of pareto-based selection. EPSOC-MO does not make explicit use of pareto dominance, selection being made according to rankings based on single objective function values, and their combination. It does not do this, however, by aggregation of the objectives.

In approach, EPSOC-MO can be seen as a hybrid of criterion-based and Pareto-based fitness assignment and selection. Fitness is determined by objective function values in turn, a criterion-based assignment. But selection, for extinction, is tempered by consideration of ranking against all objectives, an implicit Pareto-based approach.

Diversity of the population is addressed in EPSOC-MO by the same means as for its use for single objective optimisation, i.e. achieving a self-organised critical state through operations of mutation and extinction. Of the common methods (Zitzler 1999):

- Fitness sharing
- Restricted mating
- Isolation by distance
- Overspecification
- Reinitialization
- Crowding

the method of species extinction in EPSOC is closest to a form of reinitialization, though of a carefully selected section of the population.

EPSOC operates on an ordered population, ranked by fitness according to the objective function. However, for it to be used for multi-objective optimisation it would not be possible to sort the population according to two or more objectives simultaneously. EPSOC-MO, instead maintains an ordered set for each objective, with a mapping into the original population. An example for two objectives is illustrated in Figure 1.

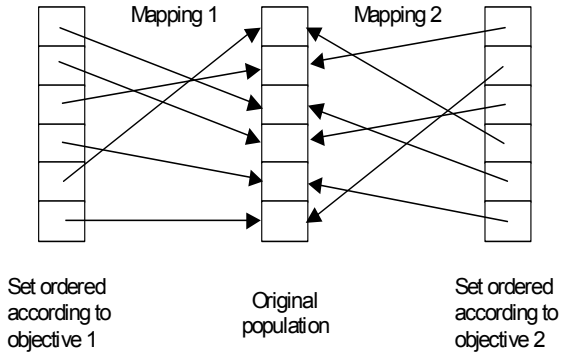


Figure 1: Ordered set mapping for multi-objective EPSOC

Referring to the outline of the EPSOC algorithm in Section 3.1, half of the *nbad* members selected come from the worst members of the first ordered set, and the remainder from the worst members of the second ordered set, *with the proviso* that those chosen from the first set not be among the elite of the second set and *vice versa*.

To implement this restriction, a set of reverse mappings, from the original population back into the ordered sets, was also maintained. Their use is illustrated in Figure 2. A member chosen from the first set, for example, would be mapped into the original population, and then into the second set. It would only be chosen for extinction if it were not in the elite of either set. For the example shown in Figure 2, the chosen member from set 1 would be rejected since it is in the elite of set 2. Similar restrictions are imposed on the choice of nearest neighbours at step 3 of the algorithm. For a problem involving more objectives, chosen members would be tested against the elite of each set in turn. The elites of each set were chosen so that their sum was the size of the elite of the single objective implementation of EPSOC**.**

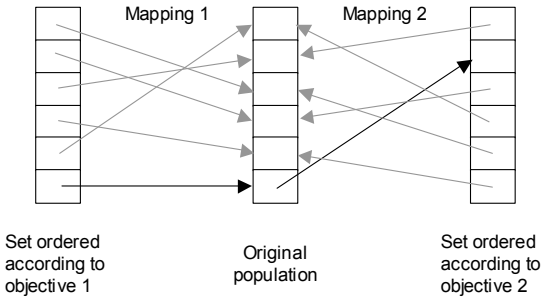


Figure 2: Use of reverse mappings in multi-objective EPSOC

The use of archives to preserve the Pareto-optimal set is commonplace in multi-objective evolutionary algorithms (MOEAs) (Zitzler 2002) and remains an area of active research (Laumanns 2002). Most algorithms use the archive as a form of "non-volatile" storage, serving only to preserve Pareto-optimal approximations. Only a few use the archive as a dynamic element in the algorithm (Knowles 1999).

In contrast, in EPSOC-MO the ranked sets play an integral role in the operation of the algorithm. These sets not only preserve the approximations to the Pareto-optimal set, but also mediate between objectives during the search. In this regard they can be considered as playing a role in the decision-making process; in effect *decision making during search* but without the interaction of the Decision Maker. In a sense, through these sets EPSOC-MO maintains multiple, "virtual" archives.

5 A Case Study: Multi Objective Radio Frequency Design

A simple experiment was constructed on a data set from a case study in radio-frequency simulation. In this problem, a ceramic bead must be designed to minimise distortion of the radiation pattern of a mobile telecommunications handset during testing (Lewis 2000). The objective function value, which is computed by a FDTD full-wave analysis of the cable structure, is a measure of transmission strength through the bead at 1 GHz. The dataset for the case study, of which isosurfaces for a particular value are shown in Figure 3, is quite complex and rich in structure. It contains 298 local minima. The

entire dataset is available as part of the Nimrod/O optimisation toolset (Abramson 2000)

A multiobjective version of this original problem can be derived by formulating a second objective for this test case. In this new scenario, not only is gain to be optimised, but the length of the optimal bead must also be minimised. This is in some sense a realistic goal, in that a shorter, more compact component might be desirable in practice. Ten runs of EPSOC-MO were performed using these objectives, each run with a population of 64 and a maximum iteration limit set to 500. The length of the bead was permitted in the range 35mm to 60mm and the range of the gain values to be minimised was from –39.85 to 48.52.

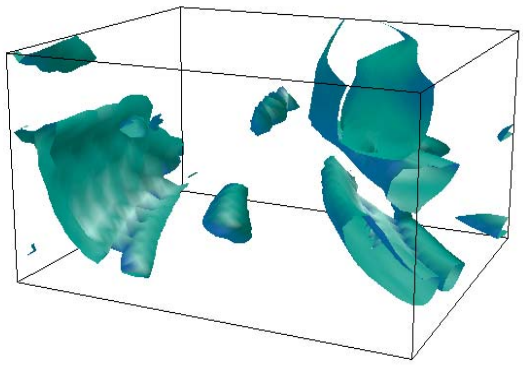


Figure 3: Isosurfaces of the gain for the ceramic bead test case

Gain and length values for the points returned at the top of the set ordered by gain for each run are shown in Table 1. The median gain for this set of points is –22.35, compared with a median of 34.99 for the whole dataset. The median length of these returned points is 48.1, compared with a median of 47.5 for the whole dataset.

Gain	Length
-12.27	50.07
-26.98	48.44
-27.55	51.77
-14.35	41.37
-20.41	41.71
-30.42	51.68
-7.73	41.34
-35.57	51.97
-10.89	42.98
-24.29	47.79

Table 1: Gain and length of "best" points from 10 runs

The entire decision space was scanned at integer intervals, and the results are plotted in objective space in Figure 4 to illustrate the distribution of solutions in objective space, and give some indication of the location of Pareto-optimal solutions. It should be noted that the test case uses these sample points and linear interpolation between them to provide objective function values, i.e. these are extremal points. The "best" returned values are highlighted. By inspection, it can be seen that the returned points approach the Pareto-optimal front of minimal gain values combined with minimal length.

The top-ranked points in the gain-ranked virtual archive in any particular run also approach the Pareto-optimal front. The top ten points from a representative run are shown in Figure 5, superimposed on the same parameter sweep data as Figure 4.

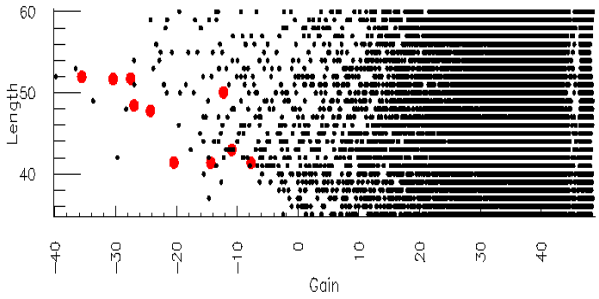


Figure 4: Multi-objective test case objective space sampling with "best" points from 10 runs

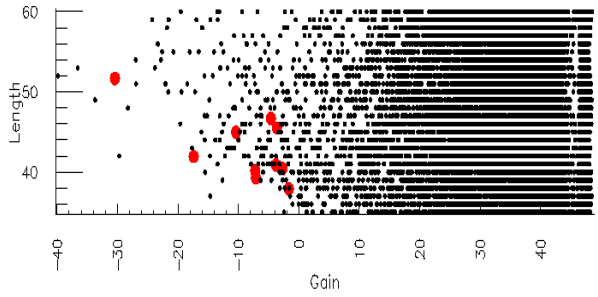


Figure 5: Multi-objective test case objective space sampling with archive points from a single run

In order to compare the effect of optimising for multiple objectives, another 10 runs were performed, the only change being the removal of the second objective function (relating to the length). The median gain of the set of points returned from these runs was –29.97, and the median length of the optimised beads was 49.0. The length of the returned beads was purely a product of the location of significant minima in the dataset, of which there are two dominant, one in a region of parameter space corresponding to a length of 52mm which also contains the global minimum, and another at length 42mm. The returned points, superimposed on the parameter sweep data, are shown in Figure 6. It can be seen that returned points are clustered at higher gain values, unlike the points in Figure 4 which are more spread along the Pareto-optimal front.

The median gain returned for the multi-objective test case is neither near that returned for the single-objective, gain-only case, nor near the median of the entire dataset.

It lies between and demonstrates EPSOC-MO is capable of returning a compromise solution, independent of an external Decision Maker supplying preference information after the search.

The median length returned for the multi-objective case lay close to the median for the entire dataset. It could be concluded that behaviour of EPSOC-MO was neutral toward the length objective, but evidence from the single-objective test case indicates this value may also be a compromise between lengths for two solutions of attractive gain in different regions of parameter space.

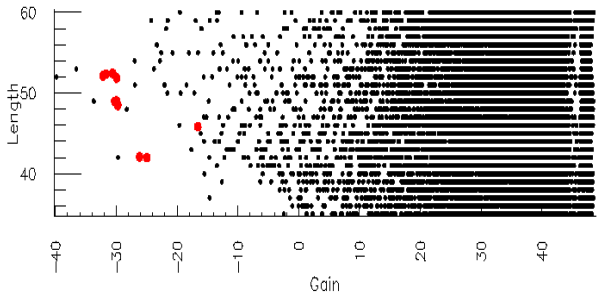


Figure 6: Test case objective space sampling with "best" points from 10 runs, optimised for gain only

6 Conclusion

An Evolutionary Programming algorithm using concepts of Self-Organised Criticality, EPSOC, has been described and a method for its use for multiobjective optimisation outlined. The algorithm uses multiple, "virtual" archives both to store optimal results and also to mediate between objectives during selection. A test of the implemented algorithm on a challenging test case with a highly nonlinear objective has demonstrated its ability to return close approximations to Pareto-optimal solutions.

These are preliminary results, and by no means exhaustive. Several questions remain as to the efficacy of EPSOC for multi-objective optimisation, not the least being the scalability of the method, but the results returned so far are quite promising. Further work is necessary to compare EPSOC-MO with other, widely-used MOEAs, and to demonstrate its behaviour on standard test problems

Bibliography

Abramson D., Lewis, A. and Peachey, T., (2000) "Nimrod/O: A Tool for Automatic Design Optimization", *The 4th International Conference on Algorithms & Architectures for Parallel Processing (ICA3PP 2000)*, Hong Kong.
http://www.csse.monash.edu.au/~davida/nimrodo/

Alander, J.T., (1995) "An Indexed Bibliography of Genetic Algorithms in Computer Aided Design", Report 94-1-CAD, Department of Information Technology and Pro-duction Economics, University of Vaasa, Finland.

Bak P. and Sneppen K. (1993) "Punctuated equilibrium and criticality in a simple model of evolution", *Phys. Rev. Let.* **71**, pp. 4083-4086.

Bak, P. (1996) "How Nature Works", Springer-Verlag, New York.

Bäck, T. (1996) "Evolutionary Algorithms in Theory and Practice", Oxford University Press, New York, NY.

Coello Coello, C.A., (1998) "An Updated Survey of Evolutionary Multiobjective Optimization Techniques" State of the Art and Future Trends", Technical Report Lania-RD-98-08, Laboratorio Nacional de Informática Avanzada (LANIA), Xalapa, Veracruz, México.

Coello Coello, C.A., (1999) "A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques", *Knowledge and Information Systems*, **1**(3), pp. 269-308.

Grefenstette, J.J., (1984) "GENESIS: A system for using genetic search procedures", In *Proceedings of the 1984 Conference on Intelligent Systems and Machines*, pp. 161-165.

Holland, J.H. (1975) "Adaptation in natural and artificial systems", University of Michigan Press, Ann Arbor, MI.

Knowles, J.D. and Corne, D.W., (1999) "The Pareto Archived Evolution Strategy: A New Baseline Algorithm for Multiobjective Optimisation", In *1999 Congress on Evolutionary Computation*, pp. 98-105, Washington, DC.

Krink, T., and Thomsen, R., (2001) "Self-Organized Criticality and Mass Extinction in Evolutionary Algorithms", *Proceedings of the 2001 Congress on Evolutionary Computation (CEC2001)*, Seoul, Korea.

Laumanns, M., Thiele, L., Deb, K. and Zitzler, E., (2002) "Archiving with guaranteed convergence and diversity in multi-objective optimization", In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 439-447.

Lewis, A., Saario, S., Abramson, D. and Lu, J., (2000) "An Application of Optimisation for Passive RF Component Design", *Conference on Electromagnetic Field Computation*, Milwaukee, June 4-7th.

Lewis, A., Abramson, D., and Peachey, T., (2003) "An Evolutionary Programming Algorithm for Automatic Engineering Design", *Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM 2003)*, Czestochowa, Poland.

Schwefel, H.-P. (1965) “Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik”, Diplomarbeit, Technische Universität Berlin.

Van Veldhuizen, D.A., (1999) “Multiobjective Evolutionary Algorithms: Classifications, Analyses and New Innovations”, PhD Thesis, Air Force Institute of Technology, Air University, USA.

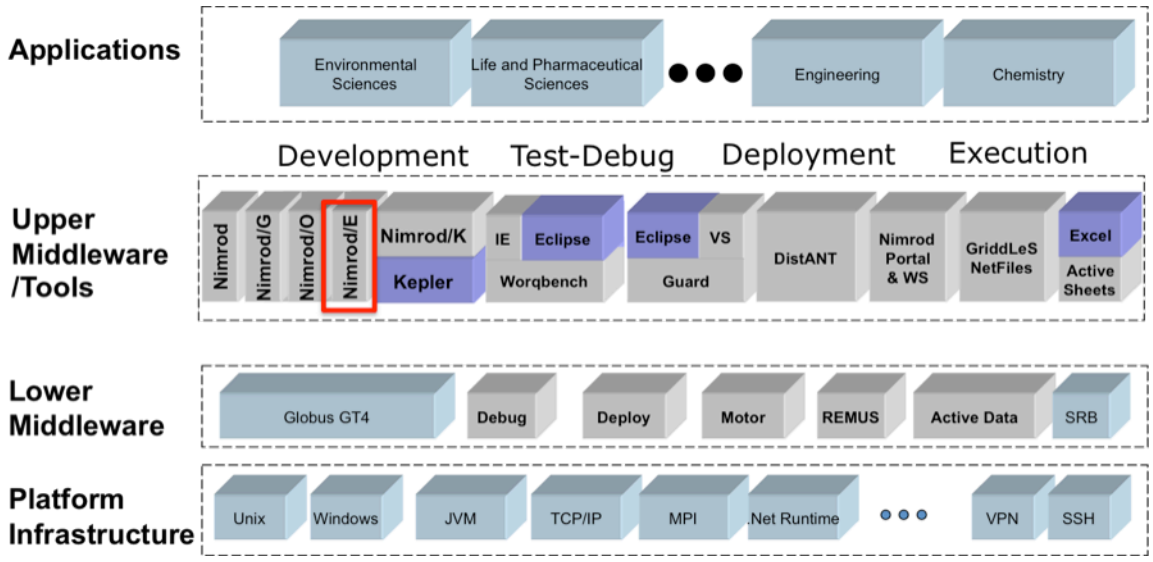
Van Veldhuizen, D.A., and Lamont, G.B., (2000) “Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art”, *Evolutionary Computation*, pp. 125-147, 8(2).

Zitzler, E., (1999) “Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications”, D.Tech.Sci. Dissertation, Swiss Federal Institute of Technology, Zurich, Switzerland.

Zitzler, E., Deb, K., and Thiele, L., (2000) “Comparison of Multiobjective Evolutionary Algorithms: Empirical Results”, *Evolutionary Computation,* pp. 173-195, 8(2).

Zitzler, E., (2002) “Evolutionary Algorithms for Multiobjective Optimization”, EUROGEN-2001, Evolutionary Methods for Design, Optimisation and Control, Barcelona, Spain.

Chapter 5: Nimrod/E and Design of Experiments



Another solution to the state space explosion experienced in parametric computing is to apply techniques from a field called "Design of Experiments (DoE)". In this paper we discuss how we have extended the basic approach used in Nimrod/G to solve very large problems, using a technique called Fractional Factorial Design.

(1) Peachey, T. C., Diamond, N. T., Abramson, D. A. Sudholt, W., Michailova, A. Amirriazi, S., "Fractional Factorial Design for Parameter Sweep Experiments using Nimrod/E", Journal of Scientific Programming, Volume 16, Numbers 2,3, 2008. 5-1 .. 5-14

Contributions

Peachey was a research associate under my supervision; Diamond provided expertise on DoE; Sudholt provided a case study in theoretical chemistry; Michailova and Amirriazi provided a case study in cardiac science.

Scientific Programming 16 (2008) 217–230
DOI 10.3233/SPR-2008-0250
IOS Press

Fractional factorial design for parameter sweep experiments using Nimrod/E

T.C. Peachey [a], N.T. Diamond [a], D.A. Abramson [a,*], W. Sudholt [b], A. Michailova [c] and S. Amirriazi [c]
[a] *Monash University, Clayton, 3800, VIC, Australia*
[b] *Institute of Organic Chemistry, University of Zurich, Winterthurerstrasse 190, CH-8057 Zurich, Switzerland*
[c] *Department of Bioengineering, University of California San Diego, and San Diego Supercomputer Centre, 9500 Gilman Drive, La Jolla, CA, USA*

Abstract. The techniques of formal experimental design and analysis are powerful tools for scientists and engineers. However, these techniques are currently underused for experiments conducted with computer models. This has motivated the incorporation of experimental design functionality into the Nimrod tool chain. Nimrod has been extensively used for exploration of the response of models to their input parameters; the addition of experimental design tools will combine the efficiency of carefully designed experiments with the power of distributed execution. This paper describes the incorporation of one type of design, the fractional factorial design, and associated analysis tools, into the Nimrod framework. The result provides a convenient environment that automates the design of an experiment, the execution of the jobs on a computational grid and the return of results, and which assists in the interpretation of those results. Several case studies are included which demonstrate various aspects of this approach.

Keywords: Experimental design, computer models, distributed execution

1. Introduction

Scientific experimentation underpins just about every aspect of modern life, from applied disciplines with practical outcomes through to our theoretical understanding of the natural world. The idea of exploring the world by controlling inputs and observing outputs is so much part of modern culture that it escapes close scrutiny in our education systems. In fact, many of us don't think about how to set up experiments in any formal way.

Early last century researchers used ad hoc approaches to varying the controllable parameters of an experiment. However, this changed in the 1920s when Fisher devised a formal and structured approach called Experimental Design [5,12,13]. Fisher showed how to select parameters in order to obtain the maximum information for a given number of experimental runs. Since then, Experimental Design has been used in a wide range of activities, and is now a large and mature theory, widely applied to scientific research and industrial processes.

With the growing power and availability of computers, physical experiments are increasingly being augmented or replaced by experiments with computer models [19]. Such models typically have many input parameters. Researchers use their knowledge of the underlying science to select the parameters that will be varied. Use of wide area computation "grids" allows many thousands of runs, even for computationally intensive models [31]. Even this, however, may be insufficient to explore all the input parameters of interest.

We perceive a need for more efficient experimental designs in the field of computer modelling. Factorial and fractional factorial designs are the most commonly used experimental designs [5], but have been underused in computer experiments. To facilitate these techniques, we have developed tools that automatically generate fractional factorial designs, perform the experiment, and then provide an analysis of the results. The tools interface with an existing tool called Nimrod/G [2], which is used to organise the execution of the model using the parallelism offered by a computational grid. The new system, known as Nimrod/E, augments the existing Nimrod tool chain [2,3,25].

The behaviour and applicability of fractional factorial designs is explored here for four case studies: two simple mathematical functions where output values and relative importance of all parameter combina-

*Corresponding author. E-mail: davida@csse.monash.edu.au.

1058-9244/08/$17.00 © 2008 – IOS Press and the authors. All rights reserved

tions can be easily determined, and two more complex realistic scientific examples.

In this paper, Section 2 expands on the use of computer models in research. The theory and application of fractional factorial design is detailed in Section 3. Nimrod is described from a user's point of view in Sections 4 and 5; the former gives a general overview of the Nimrod toolset and the latter concentrates on the provision of fractional factorial experiments using Nimrod/E. Section 6 describes the use of Nimrod/E in four case studies, chosen to demonstrate different aspect of such experiments.

2. Experiments using computer models

Computer models are now ubiquitous in many areas of research, in both physical sciences and social sciences [19]. A model that can accurately mimic a real world situation offers many advantages. It may enable experiments in situations where the real system cannot be used, where it may be physically impossible [27], too expensive [1] or unethical [34] to manipulate. The model may allow many more internal variables to be monitored and so a more detailed understanding of the behaviour. Availability of computer models has extended the possibility of experimentation to disciplines that were formally descriptive. For example, an epidemiologist may explore different drug control strategies (for example, providing public injecting rooms for injecting drug users) and may evaluate metrics such as the spread of communicable diseases like HIV/AIDS and Hepatitis [20].

A similar trend is occurring in engineering design [16]. Mathematical models are now sufficiently complex to make realistic predictions of the performance of machines and structures. This is typically cheaper than traditional prototyping, and allows a more thorough exploration of design parameters.

Computer models often have a very large number of input parameters and internal settings. Current practice with these models typically involves the experimenter identifying which parameters to vary, selecting input values for these parameters, and then generating all possible combinations of these values. In some cases, such experimentation may be required just to determine the domain over which the function operates. Each combination requires a run of the computer model. So varying even a small number of parameters can generate a huge number of runs (jobs).

The task of generating and running these jobs may be automated by software such as the Nimrod family of tools [2,3,25]. When the models are computationally intensive, then the experiment will benefit by concurrent execution of jobs on a cluster or on the distributed resources of a computational grid. Nimrod automates the distribution of jobs over available resources and the collation of results. The power of this approach is illustrated, for example, by an experiment performed at the IEEE Supercomputing Conference in 2003 involving a quantum chemistry code [31,33]. A single job with an individual parameter combination took typically about 20 minutes to run on a fast Unix machine, and the parameter sweep generated 60000 such jobs, an 800 day task for a serial machine. By using a global testbed of multi-processor clusters as a computational grid, the full experiment was completed within three days.

However, combinatorial explosion ensures that increasing the number of parameters will quickly overwhelm any computational resource. For example, just 10 parameters with four values each generate a million jobs. In the language of experimental design [5], this approach of taking the direct product of parameter values is known as a full factorial experiment. Experimental design shows how a suitably chosen subset of the full factorial experiment will be sufficient to obtain reliable results, given certain assumptions that usually apply in practice. These techniques can be dramatically parsimonious; in the 10 parameter situation above, results for a full sweep can be reconstructed from the results of just 1024 jobs, with some confidence in the results.

One difference between computer experiments and physical experiments on real world systems is the repeatability of results. Results for physical experiments have an unpredictable component, due to uncontrolled inputs, or even quantum uncertainty. Statistical analysis helps the experimenter to distinguish the signal from the noise.

For computer experiments however, unless the model is designed with random number elements, the results it produces for a given set of inputs are always the same. This gives the impression that the results are exact and have no noise component, but close inspection of most computer models refutes this. For example, many models are subject to floating point errors which may accumulate to significant values. And most models of real world systems use some discretization of the continuum and the exact choice of this effects the results. Further, models often use constant values for settings that are in fact experimental values and

hence subject to uncertainty. So the results of a computer model do contain noise and can be treated as single observations from a distribution. Thus the methods of analysis used for physical experiments apply also to computer experiments.

There is a considerable literature on the application of experimental design to computer experiments [11, 24,28]. However, these techniques have not found wide acceptance within many areas where computer experiments are practised. We believe that a significant reason for this is that they are not embedded in the design tools of choice. To remedy this deficiency, we propose to incorporate the techniques of experimental design within the Nimrod job execution framework. The first part of this project, the implementation of fractional factorial design within Nimrod, has been completed. We demonstrate the usefulness of the approach with a variety of case studies.

3. Fractional factorial experimental design

3.1. Theory

A full factorial experiment consists of a number of factors (parameters[1]), each with a number of levels (values), with all combinations of the levels being performed. The effects of each of the variables on the response, called main effects, can be determined, as well as the two-way interaction effects of each of the pairs of factors, and higher-order interaction terms.

The simplest situation is where each factor takes only two levels, a low value and a high one. Suppose the factors are $A, B, C, \ldots$ and the output of interest is ϕ. Then ϕ may always be written as

$$\begin{aligned}\phi = k &+ \{k_1 a + k_2 b + k_3 c + \cdots\} \\ &+ \{k_{12} ab + k_{13} ac + k_{23} bc + \cdots\} \\ &+ \{k_{123} abc + \cdots\} + \cdots . \end{aligned} \tag{3.1}$$

Here a is -1 when A takes the low value and $+1$ for the high. Similarly $b, c, \ldots$ are -1 or $+1$ corresponding to whether $B, C, \ldots$ are low or high. The other symbols are constant for a given output function and are called "effects", k is the mean effect, the k_i are the main effects, k_{ij} are the two-way interaction effects, and k_{ijk} etc. are higher-order interaction effects. So, for example, if $k_{13} = -3.0$ then the interaction of a and c makes a contribution of -3.0 to the output when a and c are both high or both low, and a contribution of $+3.0$ otherwise.

If there are n factors $A, B, \ldots$, then there will be 2^n effects in formula (3.1). A full factorial experiment consists of 2^n runs. Substituting the results into (3.1) provides 2^n linear equations for the effects, sufficient to evaluate them. (In the presence of noise the values will be estimates rather than exact determinations.) Assuming that the higher-order interactions contribute negligible effects, then (3.1) can be approximated by a formula with fewer terms, so a smaller set of runs may be enough to determine estimates of the remaining effects. The runs however must be appropriately chosen since different combinations of inputs can give the same linear equation for a truncated version of (3.1).

Suppose, for example, we only perform runs for which *abcde* $= +1$, these constitute one half of a full factorial experiment. Then the a term in (3.1) will always have the same sign as the *bcde* term, so the two terms are effectively combined as one, namely $(k_1 + k_{2345})a$. The estimate obtained for the primary effect of A will then include the estimate for the interaction of B, C, D and E. We say that A is "biased" by *BCDE*. Further, the estimate for *AB* will be biased by *CDE*, *AC* will be biased by *BDE*, and so on. Assuming that three-way interaction effects and higher are negligible, these biases will not be a problem. There are now half as many simultaneous linear equations but half as many effects to be determined. For this example, *ABCDE* is called a "word in the defining contrast"; each additional word in the defining contrast halves the number of runs, resulting in a "fractional factorial design".

To create a fractional factorial experiment the designer must first decide which effects are required to be estimated and which can be considered negligible. Defining contrasts are then chosen that allow the former to be biased by the latter. More fully, the effects are divided into three classes:

- Primary effects: effects for which estimates are required.
- Secondary effects: effects for which estimates are not necessarily required, but are not allowed to bias the estimates of the primary effects.
- Other effects: effects for which estimates are not necessarily required, and are allowed to bias the estimates of the primary effects.

[1] We adopt Fisher's nomenclature from this point on, referring to function "parameters" as "factors".

For a full factorial design, the primary effects would include all effects, main effects, two-factor interactions, three-factor interactions, and so on. By restricting the primary effects, the size of the design (i.e., the number of runs) can be reduced.

The most important fractional factorial designs are described using a special shorthand terminology developed by Box [5]. These are referred to as "resolution III", "resolution IV" and "resolution V". In resolution III designs, the primary effects set is the set of all main effects, while the secondary effects set is empty. Hence, main effects might be biased by two-factor interaction effects. In resolution IV designs, the primary effects set is the set of all main effects, and the secondary effects set is the set of all two-factor interaction effects. Main effects may only be biased by three and higher-order interactions, but not two-factor interactions, and the two-factor interaction effects may not be estimable. In resolution V designs, the primary effects set is the set of all main effects and the set of all two-factor interaction effects, and the secondary effects set is empty. With these designs all main effects and two-factor interaction effects can be estimated, biased by three-factor and higher order interaction effects. Designs with higher resolution than V can be produced by extending this pattern, but these are less commonly used since three-factor interactions are usually small.

Fractional factorial designs are easier to design and analyse for two-level factors than for factors with more than two levels. However, a 2^n level factor is equivalent to n two-level "pseudo-factors", and hence the theory and practice for two-level designs can be easily extended to designs with factors with more than two levels. For direct use of pseudo-factors, the number of levels needs to be a power of two. For example if a factor a has four levels, then these are indexed by two pseudo-factors, say *A1* and *A2*, each corresponding to one bit in the index. While it is convenient to design the fractional factorial experiment in terms of the pseudo-factors, the analysis is best conducted by using orthogonal polynomials for the main effects, and for the interaction effects for designs of resolution V. See Draper and Smith [10] for more details.

3.2. Analysis

An experimental design will provide a list of runs to be performed; each run assigns a value for each input factor. (In computer modelling the runs are usually called "jobs".) Once these jobs have run and each produced a numerical result, then the estimates of effects, the coefficients in Eq. (3.1), can be obtained as described above. These results can be used in various ways.

1. The relative sizes of the effects may be of scientific interest. For example, very small values for the main effect and interactions of a particular factor suggest that the factor may be irrelevant in the model. If two factors have significant main effects but a small interaction, then the factors are making independent contributions to the result. The experimenter with knowledge of the science is best placed to interpret and make use of such results. In particular, disagreement between the effect estimates and what would be expected from the science may indicate problems with the computational model.
2. Results for effects may suggest larger experiments. For example if, in a resolution V experiment, the two-way interactions *AB* and *AC* are found to be large, it suggests that the three-way interaction *ABC* may not be negligible. The experimenter may then add this to the primary effects and design a new experiment that will estimate this effect.
3. An experiment with a small number of levels per factor may be performed as a preliminary to one with more levels. The factors found most significant in the preliminary experiment might warrant more levels in the later one.
4. When many levels are taken for a factor, the effect corresponding to the least significant bit gives an indication of the smoothness of the response surface.
5. Knowledge of the relative sizes of effects can assist in subsequent optimization [8] procedures. For example, if factors have negligible effect then it may pay to first perform a search that ignores those factors, as reduced dimensionality assists some search algorithms. Again, if the factors fall into two groups with no interaction between groups, the output will be the sum of two independent parts. Then optimizing each part separately is possible and may be advantaged by the reduced dimensions.
6. Estimates of the effects can be used in Eq. (3.1) to compute the results of a full sweep. Where there are many factors, this may be the only practical way to produce such results.
7. The results produced by predictions based on estimates of effects are a form of data smoothing as higher order interaction effects are removed. The

results may be used to reduce noise or to detect errors in the results.

It follows from our discussion in Section 2 that the estimates of effects should be treated as containing some random noise. How do we decide whether the values obtained are significant or just noise? The standard methods of analysis assume that the noise components are normally distributed with a zero mean. On this basis, two graphical methods have been devised that display all the estimated effects and facilitate judgement of their significance: the Daniel plot [9] and the Lenth plot [17].

A Daniel plot shows the effects in numerical order on probability paper. Under the hypothesis that none of the factors affects the result, the plotted points will fall in a straight line. Typically, most higher-order effects are negligible, and so many points do fall on a line. Those points that clearly do not lie in the line are probably significant. An example is shown in Fig. 7. It clearly shows that single effects A, B and the interaction *AB* are significant. However, the assessment of these plots is somewhat subjective.

An alternative display is the Lenth plot. Here the estimates of effects are plotted vertically as in Fig. 6, in order from largest to smallest in absolute size. Horizontal lines indicate the significance of the effects. The inner dashed lines are known as ME (margin of error) lines. Effects that are in fact zero have a probability of 95% of falling within these lines. The outer solid lines, SME (simultaneous margin of error), apply to all effect estimates as a group. If all effects are zero there is a probability of 5% of one estimate falling outside these lines. So if a point falls outside the SME lines then the effect is probably significant, while if the point falls within the ME lines then there is no strong evidence of significance, non-zero values may just be noise.

Most computational models of course produce many numerical results. Once a fractional factorial experiment has been designed and run, a separate analysis may be produced for each result. The effect estimates may be very different in the separate analyses. For example, one output may be strongly affected by factors A, B and C while another by factors D, E and F. Such information may be significant in understanding the computational model.

The fractional factorial methodology may give the impression of a simple linear approach: pose the question, design the experiment, take the measurements, and interpret the results. However, real problems are rarely as simple as that, as Box [6] has observed. Box describes two categories of research: the initial "discovery" phase that generates hypotheses, and the later "testing" phase where they are confirmed or rejected. As Box explains, there are no recipes for the discovery phase of research. It depends on both implicit and explicit knowledge of the researchers, on judgement and on lateral thinking. It is typically a messy iterative procedure. We see fractional factorial experiments as a useful component in this iterative process.

4. The Nimrod toolset

Nimrod is a family of tools that facilitate parametric studies with computational models, that is, the exploration of the effect of varying the input parameters on the results produced. Nimrod has been extensively used for this purpose [7].

Nimrod/G [2] was designed to assist engineers and scientists in performing such studies, using concurrent execution on a cluster of processors or the resources of a computational grid. The user prepares a "plan file" which specifies the experiment; Fig. 1 shows an example. The first section specifies the parameters (factors) and the values assigned (levels). The "task" section specifies how the computational model is to be executed using distributed execution. Here the executable "compModel" is copied to a remote node together with a generic input file, the input file is "substituted" to insert current values for the parameters and the model then executed to produce results in a file called "results". This is then copied back to the root node and given an extension "$jobname", which is a unique Nimrod identifier.

A plan file is expanded into a "run file" which lists the appropriate parameter combinations required for execution. The Nimrod/G core will run on a processor called the "root node" whereas individual jobs are executed on "remote nodes". Nimrod/G handles the file transfers required to the remote nodes, execution of computational tasks and transfer of results back to the root node. The number of concurrent jobs is limited only by the number of processors available. Thus the user may achieve high concurrency without modifying the executables and without concern for grid specific details. A commercial version, enFuzion [4], has been used in financial modelling, bio-informatics, graphics rendering and modelling of power grids.

Nimrod/O [25] optimizes the numerical outputs of computational models. It provides a range of optimization algorithms and leverages Nimrod/G or enFuzion to perform batches of concurrent evaluations. The user

```
parameter temp float range from 200 to 300 points 2
parameter pressure float range from 5000 to 6000 points 4
parameter concent float range from 0.002 to 0.005 points 2
parameter material text select anyof "Fe" "Al"
task main
   copy compModel node:compModel
   copy inputFile.skel node:inputFile.skel
   node:substitute inputFile.skel inputFile
   node:execute ./compModel < inputFile > results
   copy node:results results.$jobname
endtask
```

Fig. 1. Plan file for a Nimrod/G experiment.

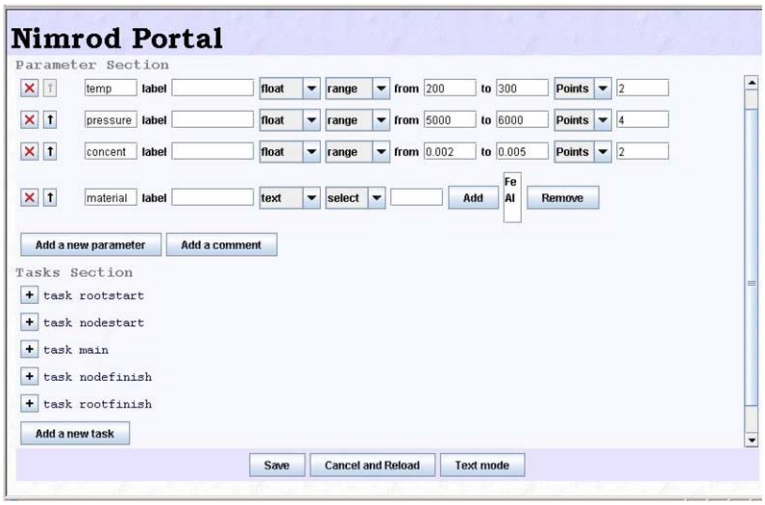


Fig. 2. Nimrod portal planning page.

prepares a "schedule file", which, like the Nimrod/G plan file, specifies the parameters, their ranges and the tasks required for execution of the model. But it also specifies the optimization algorithms to be used and the settings for the algorithms. Nimrod/O performs multiple searches for an algorithm and multiple algorithms, all in parallel.

Although all of the Nimrod tools can be used by experienced users in a "command line" mode, most users prefer the interactive Nimrod Portal [22]. The Nimrod Portal uses drop down menus to design and run an experiment and to select computational resources. Such resources may be added or removed as the experiment proceeds. The Portal allows a user to manage their computational testbed without resorting to the detailed and complex command line tools used by the underlying grid middleware.

Figure 2 shows a page from the Portal where the user plans the experiment. When the user has selected parameters and tasks, the "save" button will create a Nimrod plan file.

Whilst a Nimrod user could apply the experimental design techniques discussed in Section 3 by consulting published tables of designs, or experimental design programs, to prepare a run file, we have integrated the techniques into the Nimrod tool chain. The result is convenient and less prone to error, and is the motivation for the development of Nimrod/E.

5. The design and architecture of Nimrod/E

A fractional factorial experiment within the Nimrod framework is a four-stage process, as shown in Fig. 3. Given the factors to be controlled and the levels used for those factors, the first stage is the production of an experimental design. For this purpose, we have implemented the RSSEF algorithm (discussed in Section 5.2) in a program called *nimrodFracDes*. The program produces a Nimrod/G run file, which specifies which jobs are required. When Nimrod/G has completed these jobs and assembled the corresponding result files, a third application called *nimrodFracAn* analyses the results, producing results in a graphical form known as Daniel plots and Lenth plots. The

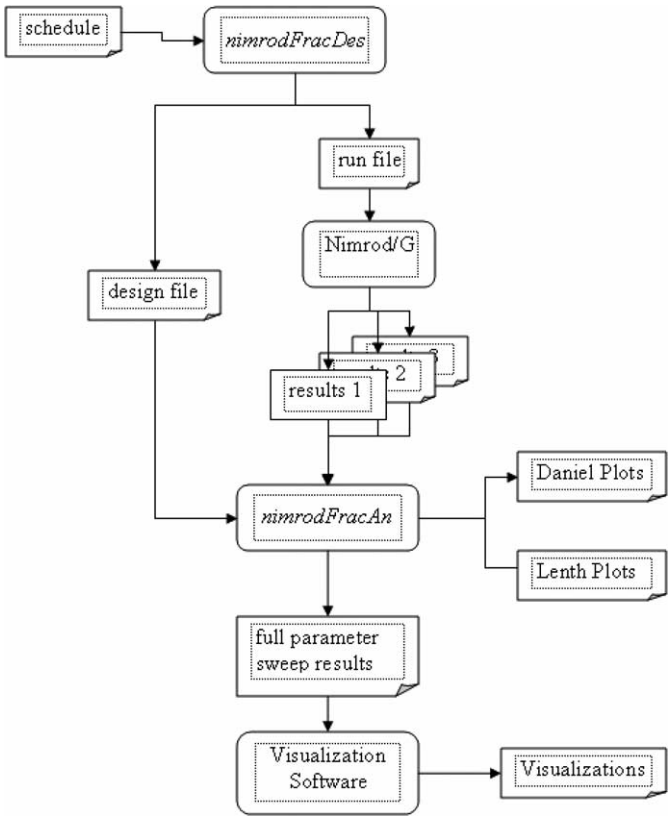


Fig. 3. Workflow for a Nimrod/E fractional factorial experiment.

application also uses these effect values to generate the results of a full factorial experiment. It thus fills in the results for the runs that have not been performed. The results are then in a form suitable for visualization software such as OpenDX [23]. However, we have not incorporated a visualization component in the present system.

5.1. The experiment schedule

The experimenter's input may be confined to the preparation of a "schedule" file. Such a file contains four sections, as shown in the example of Fig. 4. The first two sections are the standard format used in Nimrod plan files as described above. The following "design" section specifies either the resolution of the design, or else directly specifies the primary and secondary effects. Here the experiment will have resolution V.

The final (optional) section controls the analysis to be done after the fractional factorial runs are complete. For this experiment there will be three separate analyses performed, corresponding to three different output values found in the output file. Nimrod/E has a record of which values for $jobname correspond to which runs and hence can compute the estimates of the effects for each of the three results.

5.2. Creation of the factorial design

Computer experiments often involve many factors, more than traditionally handled in physical experiments. Published tables of experimental designs do not extend to more than 20 factors for resolution V. There are several published algorithms for fractional factorial design [14,15,18]. Of these, the only *general purpose* one that can handle many factors is the SEF algorithm of Liao and Iyer [18], which can cope with 80 factors at two levels each, for a resolution V design. This may seem sufficient for practical purposes. However, if more than two levels are required the number of factors is significantly reduced. For example, with eight levels per factor only 14 factors can be handled. Consequently, we have developed a variant of that algorithm, called RSSEF [26], which can handle 130 two-level factors or 18 eight-level factors.

```
parameter a float range from -1 to 1 points 2
parameter b float range from -1 to 1 points 2
parameter c float range from -1 to 1 points 2
parameter d float range from -1 to 1 points 2
parameter e float range from -1 to 1 points 2

task main
   copy testFunction node:testFunction
   node:execute ./testFunction $a $b $c $d $e > results
   copy node:results results.$jobname
endtask

design
  resolution 5
enddesign

results "NoNoise" in file "results" line 1 field 1
results "Noise30%" in file "results" line 2 field 1
results "Noise100%" in file "results" line 3 field 1
```

Fig. 4. Schedule file for Experiment 1.

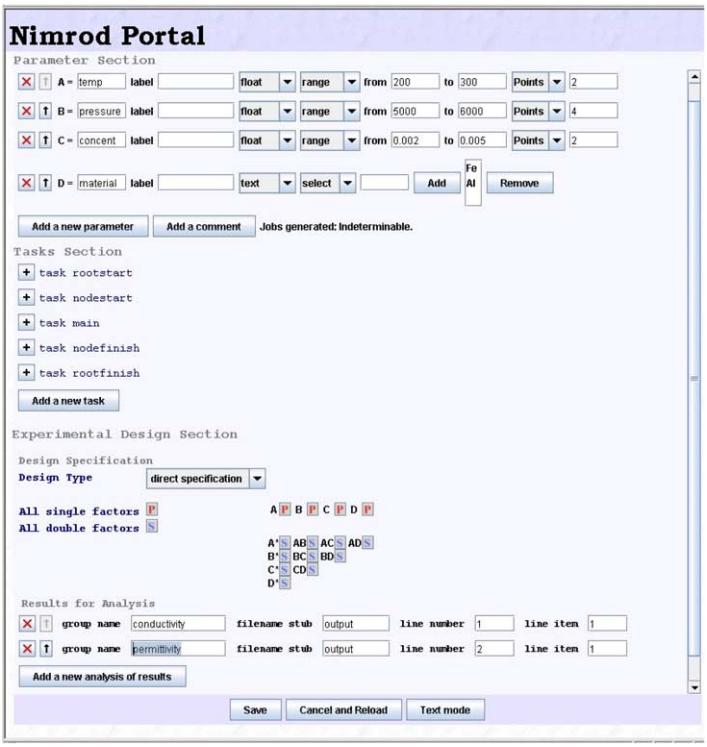


Fig. 5. Screenshot of Nimrod portal experimental design planning page.

5.3. *The Nimrod/E portal interface*

Access to Nimrod/E has been incorporated into the Nimrod Portal. If an experiment is specified as Experimental Design then the page for editing a schedule contains extra functionality for specifying the design and the results for analysis. Figure 5 shows this page, with the four sections corresponding to the four parts of an experiment schedule. In the "Design Specification" section the user may select the resolution required for the experiment. Alternatively each main effect and each two-way effect may be individually specified as either primary (P) secondary (S) or other (blank) thus allowing finer control of the design. There is an alternative text mode interface that permits specification where a list of primary and secondary effects is typed. This mode permits three-way and higher interaction effects.

The final section "Results for Analysis" is used to specify where, among the results of the runs, the numbers for analysis will be found. In this case each run is producing a file called "output" where these numbers are written. Two analyses will be performed; one on the first item in the first line, and another on the first item in the second line.

When this design is saved, the Portal will use *nimrodFracDes* to compute the fractional factorial design and the Nimrod/G run file. If the user has also selected computational resources then no further user input is required. The Portal will launch Nimrod/G to perform the runs and, when the runs are complete, will apply *nimrodFracAn* to produce the analyses.

6. Case studies

This section illustrates the use of Nimrod/E and reports the results of four example experiments. The first two use known output functions and are included to demonstrate the principles of experimental design and analysis. The third experiment uses a computational model of heart muscle cells; the analysis was used to gain insight into the nature of the objective calculated. The final experiment is a re-analysis of data obtained from an earlier large computational chemistry experiment and demonstrates another use for Nimrod/E.

6.1. Experiment 1: A known quadratic response

This section illustrates the use of fractional factorial design using a simple model where the response function is known. This function is

$$\begin{aligned}\phi(A, B, C, D, E) &= 2A - B + 0.1D + AB \\ &\quad - 0.2AE + \sin \pi AC + kN, \end{aligned} \tag{6.1}$$

where N is a random number selected from a rectangular distribution over an interval $[0, 0.01]$, and k will be chosen later. We use two levels, -1 and $+1$, for each parameter A, B, C, D, E.

In many computational models the input parameters vary in importance, with some making negligible contributions to the results. The response function (6.1) reflects this with D having only a small effect; E has a small effect also, but only as an interaction with A. Although C is significant within the function, it appears only in the sine term which is zero for the levels used, so it has no effect in this experiment. This term is chosen to illustrate the influence of the chosen levels.

The random number term, kN, has been included to illustrate this effect of noise on estimates of the effects. We perform experiments with various values of k, namely 0, 0.3 and 1.0, to vary the noise component. If $k = 0$, then the function is a linear combination of single parameters and two-combination products, so we should expect that a resolution V experiment will give an exact reconstruction of the full factorial experiment.

Figure 4 shows the schedule used for this experiment, prepared by interaction with the Nimrod Portal. Nimrod/E used this schedule to compute a resolution V design, which generated 16 runs as compared to $2^5 = 32$ for a full parameter sweep. The results are presented in graphical form in the Lenth plots of Fig. 6.

With no noise the effect estimates exactly equal the coefficients of formula (6.1), so that the results for the 16 jobs are sufficient to reconstruct all 32 results for these levels. With 30% noise the small effect of the 0.1D term is obscured; the D effect is in the zone of uncertainty between the ME and SME lines. With 100% noise both the *AE* and D effects are within the ME lines and so cannot be deemed as active.

The alternative presentation of effects using Daniel plots is shown in Fig. 7. To save space only the 100% noise case is shown. The points representing the effects A, B, and *AB* clearly deviate from the straight line and may thus be deemed significant.

This experiment illustrates item 1 in Section 3.2, how fractional factorial design can identify the relative importance of factors and factor interactions, and how these become obscured as the amount of noise increases.

6.2. Experiment 2: A more complex known response

The next experiment explores the effect of the choice of resolution, again using a model with a known output function. In this experiment we concentrate on the reconstruction of the full factorial experiment from the results of fractional factorial ones, item 3.2.6 in our list of applications.

In this example, the output function $\phi(A_1, A_2, A_3, A_4, A_5, A_6)$ is defined recursively by $\phi_1 = A_1, \phi_i = \sqrt{100.0 + \phi_{i-1} + iA_i}$ for $i = 2, 3, \ldots, 6$ and $\phi = \phi_6$. We use two levels, 0 and 1, for each input factor. Nimrod/E was used to design three fractional-factorial experiments, with Resolution III, IV, and V, respectively. In each case the estimates of the effects were

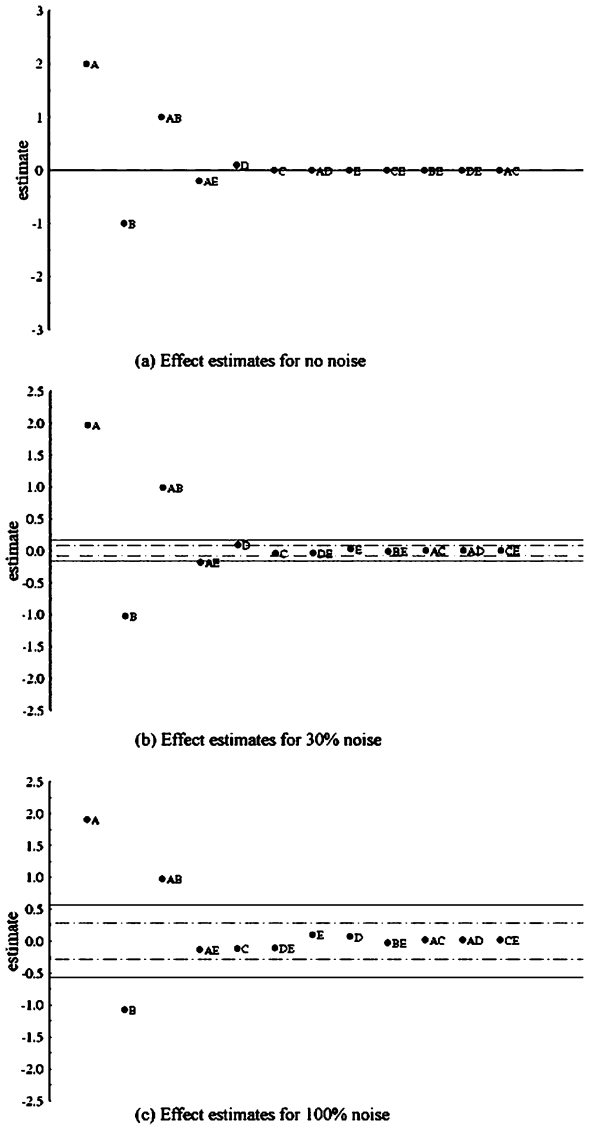


Fig. 6. Lenth plots for Experiment 1.

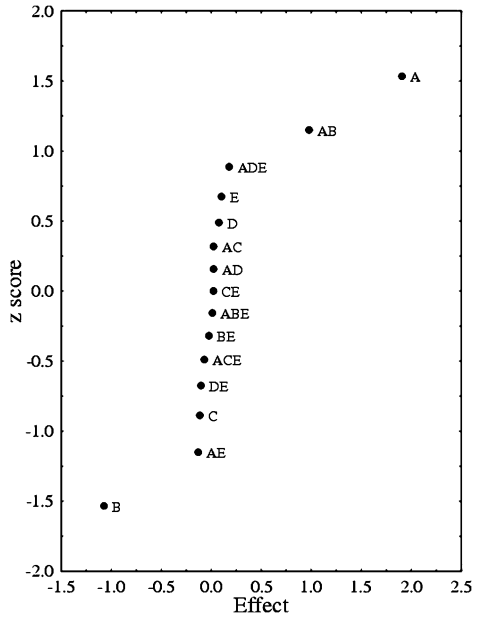


Fig. 7. Daniel plot for Experiment 1 for 100% noise data.

Table 1
Reconstruction of a full sweep from fractional-factorial Experiment 2

Resolution	No. runs	SSD	SSND
III	8	1.26939	0.0111786
IV	16	3.89334e–3	3.43098e–05
V	32	9.46657e–12	8.31543e–14

used to construct results for a full parameter sweep. To assess the discrepancy between these estimated values E_i and the genuine results of a full parameter sweep, R_i, the usual statistic $SSD = \sum(E_i - R_i)^2$ was computed. As this statistic fails to indicate the error relative to the true values, a normalized statistic $SSND = \sum[(E_i - R_i)/R_i]^2$ was also computed. Table 1 shows the results for all of these experiments.

Resolution V produced results very close to the true values, indicating that the effects of interactions of more than two factors are small. The results illustrate the trade-off between the number of runs and the accuracy of the predictions; as the resolution is decreased, the number of runs reduces but the accuracy of the predictions also falls, but is still high. The close agreement between predictions and true values demonstrates a major justification for the fractional factorial approach.

6.3. Experiment 3: Ionic flux in a cardiac model

This experiment concerns a mathematical model of excitation-contraction coupling in the rabbit ventricular muscle cells [30]. In the model a system of ordinary differential equations is used to describe the intracellular ionic fluxes (for Ca^{2+} and Na^{+}). The model incorporates many features of cell structure and is able to reproduce experimental data using parameters based on measurements in rabbit cells. In this experiment we extended the model to incorporate Mg^{2+}-nucleotide regulation [21]. The experiment involves validation of the updated model by varying nine input parameters (see legend in Fig. 8) so that the predicted I_{NaK}, I_{Ca}, I_{pCa} and I_{SERCA} currents are as close as possible to

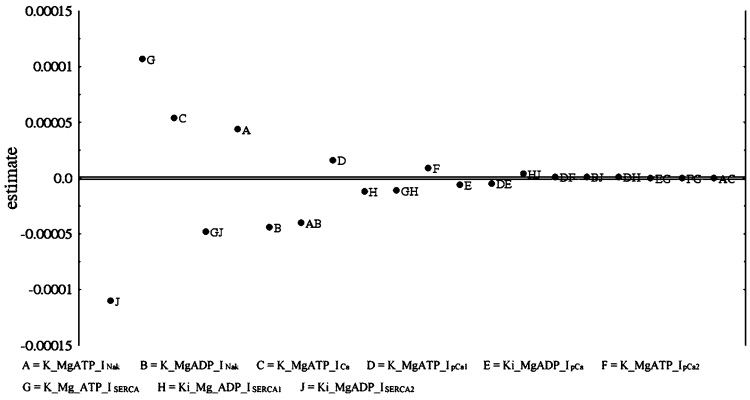


Fig. 8. Lenth plot for Experiment 3.

the experimentally measured. This was phrased as a optimization problem using Nimrod/O, minimizing the sum of the moduli of the differences between computed metabolic factors (I_{NaK}, I_{Ca}, I_{pCa}, I_{SERCA}) and experimental values.

The code is implemented in Matlab, and each computation takes about 10 minutes on a high end workstation. Prior to running Nimrod/O, we decided that a preliminary parameter sweep with Nimrod/G would indicate the broad features of the objective landscape and this might lead to more efficient searches in Nimrod/O. However, with nine parameters a full sweep becomes a large undertaking, so we performed a Nimrod/E experiment instead.

Here we denote the metabolic factor inputs as $A, B, \ldots, G, H$ and J (by convention I is not used to denote factors in experimental design). Two values were chosen for each of these factors which were judged from experience to span the realistic range. A resolution V experiment was designed and executed.

Figure 8 shows the Lenth plot for the most significant effects on the basis of the resolution V experiment. Clearly, all nine factors had a significant effect. Of the two-way interactions, the significant ones are AB, DE, DF, GJ, GH and HJ. Other combinations cannot be distinguished from noise. This prompted a closer look at the model which explained this result. The factors fall into four groups $\{A, B\}$, $\{C\}$, $\{D, E, F\}$ and $\{G, H, J\}$. It turned out that only the first group affects the first term in the objective, only the second affects the second term, and so on. Thus we would expect no interactions between factors in different groups. Whilst the result was obvious with hindsight, the work demonstrated the power of a tool like Nimrod/E – we had not expected this result, and were treating the computational model as a black box. Notably, Nimrod/E's analysis discovered the lack of coupling between certain factors without detailed knowledge of the code. The reasons for this lack of coupling only became obvious to us once we looked *inside* the computation.

The result also had significant implications for the Nimrod/O run that was to follow. As discussed in item 3.2.5, a more efficient search was possible by separately optimizing the four components of the output.

6.4. *Experiment 4: A quantum model for complex molecules*

This experiment concerns quantum chemical models – based on the Schrödinger equation – of assemblies of atoms as found in large biomolecules. The problem with using a pure quantum mechanical approach is that these methods are extremely computationally expensive, and thus can normally only be applied to relatively small molecules. Hybrid quantum mechanics–molecular mechanics (QM/MM) models, on the other hand, just describe a small, "active" region by accurate quantum techniques, while the surrounding larger, "inactive" region is treated with more approximate classical force fields. As a result, they tend to be faster than a full quantum chemistry computation. Unfortunately, the two involved physical concepts are so different that they cannot be easily combined. One approach to solve this combination is to saturate each of the quantum chemical bonds between both regions with a single parameterized "capping atom", instead of with the bulk of the large molecule. This capping atom is not real, but can be adjusted by a hypothetical pseudopotential function [35], as shown in Fig. 9.

This experiment explores such a capping potential for a carbon–carbon single bond, using the "group dif-

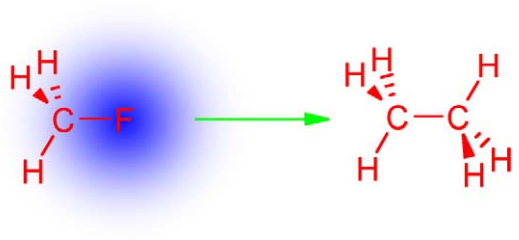


Fig. 9. Replacement of part of the structure with a capping atom (F) and a pseudopotential.

ference potential" (GDP) approach of Sudholt et al. [31–33]. In this model, ethane (CH_3–CH_3) is used for the parameterization. One of its methyl groups (CH_3) is substituted by an isoelectronic fluorine atom (F) and a pseudopotential of the form

$$U_{\mathit{eff}}(r) = A\exp(-Cr^2) + B\exp(-Dr^2), \quad (6.2)$$

where r is the radius. Here parameters A, B, C, D are to be determined that will provide the best fit between the model and target properties of ethane and the methyl radical. This is assessed by a normalized least squares expression. The quantum chemistry code GAMESS [29] was used to compute the model properties used in this comparison, each task taking about 20 minutes of computation time.

The experiment used the results of the very large sweep over the A, B, C, D parameter space described in [31]. The results showed a huge variation in the size of the output results, from ∼500 to over 10^{11}, so we decided to analyse the logarithms of these values rather than the direct values.

With four factors, a resolution V fractional factorial design is just a full sweep. A resolution IV design does give a reduction of 50% in the number of runs but biases the estimates of effects. We used Nimrod/E to perform a resolution IV experiment with two levels per parameter. Figure 10 is a Daniel plot for the results. It shows that the effect largest in absolute size is the four-factor interaction *ABCD*, which can be understood from the mathematical form of the pseudopotential. This indicates that the usual result, that higher order effects are smaller than lower order ones, fails in this case. So a reconstruction of the full sweep using results of the resolution IV experiment cannot be expected to give accurate results here. This case study illustrates the robust nature of the methodology which indicates when a full sweep is required. Nothing is lost by performing a fractional factorial experiment first as the full sweep can be designed to perform only the missing runs.

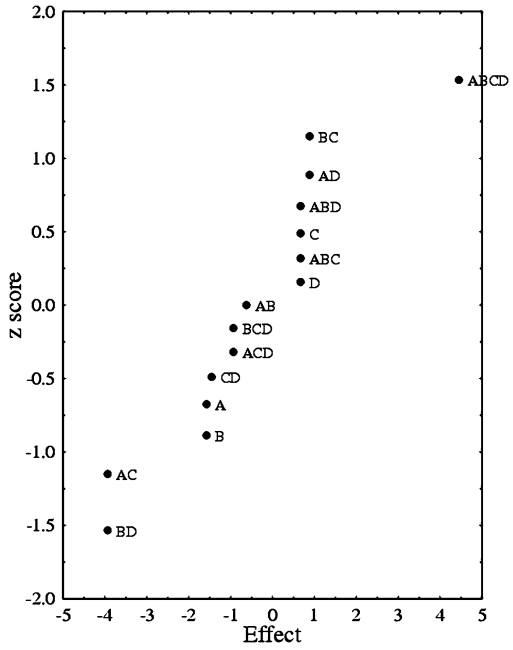


Fig. 10. Daniel plot for Experiment 4.

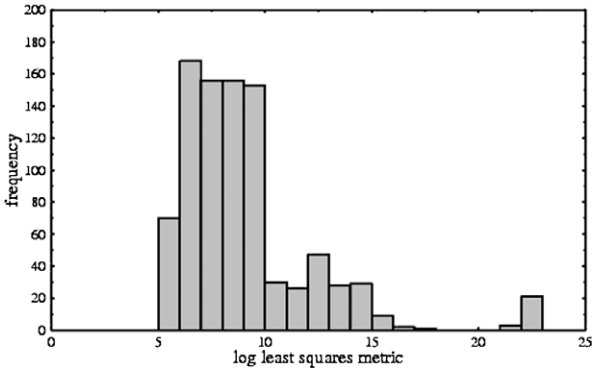


Fig. 11. Histogram of results for Experiment 4.

There was however one application of fractional factorial analysis that was useful with this data. We decided to use a design with more than two levels per factor. In the original data the number of levels were not powers of two, so we extracted a subset using 8 levels for each factor A and B, and 4 levels for each of C and D. This produced a data set of 1024 values. A histogram of these values, Fig. 11, revealed a gap in the distribution. A small proportion of results were very large, over 21, with nothing in the range 18–21. This gap was not evident in the full dataset which showed a contiguous distribution. This naturally raised the question as to whether the outlying results were valid, or a symptom of a failure of the model. To explore this we

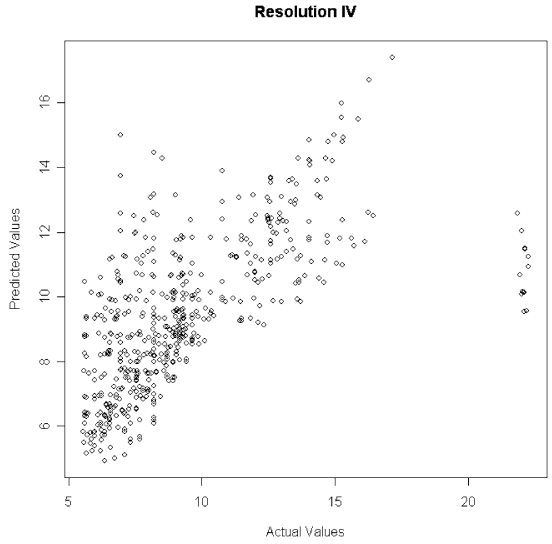


Fig. 12. Correlation between predicted and actual values for Experiment 4.

used reconstruction from a fractional experiment as a smoothing filter on the data, as discussed in Section 3.2 item 7.

Nimrod/E was used to design a resolution IV experiment, run the experiment, and produce predicted values. In Fig. 12 the predicted values are graphed against the actual values on a scatter plot. The correlation is not high, but sufficient to indicate that all the outliers, the points on the right side, are aberrant, they reveal a discontinuity in the response function. A likely explanation is failures in the GAMESS calculations that were not flagged to the output. This example illustrates the application of fractional factorial design to assess the validity of data produced by computational experiments. Here we see the cyclical nature of experimentation, results feeding back into the model which may then be adjusted and produce the next round of results.

7. Conclusions and further work

This paper illustrates the use of fractional factorial methods in design of computer experiments. Fractional factorial designs typically offer a large saving in the number of jobs required to explore a parameter space, but also provide a robust way of indicating when a full sweep might be required. Furthermore they provide tools for the experimenter to assess the relative importance of the various parameters and their interactions and offer insights into the computational model.

To facilitate these methods we have developed a tool, Nimrod/E, which automates the design of the experiment, interfaces with Nimrod/G for the execution of the experiment, and then provides tools for the final analysis. A complete project may be controlled from the Nimrod Portal. This offers a considerable advantage over previous approaches that require operator intervention to supply a design appropriate to the situation.

We have illustrated the effectiveness of Nimrod/E with four case studies. Two of these were hypothetical examples used to demonstrate the basic functions, but two were real world scientific experiments. Importantly, Nimrod/E produced useful results in both of the real world studies. In the cardiac modelling work, Nimrod/E detected the structure of the underlying computational model without any specific details, and allowed us to make an improvement in the overall performance of a subsequent optimization run. In the quantum chemistry work we showed the necessity of a complete parameter sweep and that it was not possible to perform a simpler and cheaper experiment than the one we did. Further, a comparison of the fractional factorial predictions with results of the full sweep was used to investigate questions raised by some of the model outputs.

We see the implementation of fractional factorial designs as the first stage of a continuing project. Other experimental designs offer advantages, for example: empirical kriging methodology, Latin hypercube designs and maximum entropy designs [11]. We hope to include them in the final suite of Nimrod/E tools.

Acknowledgements

Nimrod/E was developed under funding provided by the Australian Research Council as part of the Australian Centre for Complex Systems (ACCS). The cardiac modelling work was supported in part by the U. S. National Science Foundation, the California Institute for Telecommunication and Information Technology, and the National Biomedical Computation Resource. The quantum chemistry application was supported in part by the German Academic Exchange Service (DAAD) and the University of California, San Diego.

References

[1] D.A. Abramson, M. Cope and R. McKenzie, Modelling photochemical pollution using parallel and distributed computing platforms, in: *Proceedings of PARLE-94*, Athens, Greece, 1994, pp. 478–489.

[2] D. Abramson, J. Giddy and L. Kotler, High performance parametric modeling with Nimrod/G: Killer application for the global grid?, in: *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[3] D. Abramson, P. Roe, L. Kotler and D. Mather, ActiveSheets: Super-Computing with Spreadsheets, *2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference,* Seattle, USA, April 22–26, 2001, pp. 110–115.

[4] Axceleon, http://www.axceleon.com/, accessed 20 August 2007.

[5] G.E. Box, W.G. Hunter and J.S. Hunter, *Statistics for Experimenters*, Wiley, 1978.

[6] G.E. Box, Statistics and quality improvement, *Journal of the Royal Statistical Society A (Statistics in Society)* **157** (1994) 209–229.

[7] Collaborative Projects using Nimrod Tools, http://message.infotech.monash.edu.au/Nimrod/Projects, accessed 20 August 2007.

[8] R.P. Brent, *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, 1973.

[9] C. Daniel, *Applications of Statistics to Industrial Experimentation*, Wiley, 1976.

[10] N. Draper and H. Smith, *Applied Regression Analysis*, 2nd edition, Wiley, 1981.

[11] K.-T. Fang, R. Li and A. Sudjianto, *Design and Modelling for Computer Experiments*, Chapman and Hall/CRC, 2006.

[12] R.A. Fisher, *Statistical Methods for Research Workers*, Oliver and Boyd, 1925.

[13] R.A. Fisher, *The Design of Experiments*, Oliver and Boyd, 1935.

[14] M.F. Franklin and R.A. Bailey, Selection of defining contrasts and confounding effects in two-level experiments, *Applied Statistics* **26** (1977), 321–326.

[15] A. Fries and W. G. Hunter, Minimum aberration 2^{k-p} designs, *Technometrics* **22** (1980), 601–608.

[16] R. Jones, D. Peng, P. Chaperon, M. Tan, D. Abramson and T. Peachey, Structural optimization with damage tolerance constraints, *Journal of Theoretical and Applied Fracture Mechanics* **43** (2005), 133–155.

[17] R.V. Lenth, Quick and easy analysis of unreplicated factorials, *Technometrics* **31** (1989), 469–473.

[18] C.T. Liao and H.K. Iyer, A stochastic algorithm for selecting [sic] of defining contrasts in two-level experiments, *Biometrical Journal* **6** (1999), 671–678.

[19] The London e-science centre, http://www.lesc.ic.ac.uk/support/projects.html, accessed 26 July 2007.

[20] D. Mather and N. Crofts, A computer model of the spread of hepatitis C virus among injecting drug users, *European Journal of Epidemiology* **15** (1999), 5–10.

[21] A. Michailova, W. Lorentz and A. McCulloch, Modelling transmural heterogeneity of KATP current in rabbit ventricular myocytes, *AJP-Cell Physiology* **293** (2007), C542–C557.

[22] Nimrod Portal, http://www.csse.monash.edu.au/∼davida/nimrod/portal.htm, accessed 20 August 2007.

[23] OpenDX, http://www.opendx.org/about.html, accessed 26 July 2007.

[24] A.B. Owen, Latin supercube sampling for very high-dimensional simulations, *ACM Transactions on Modeling and Computer Simulation* **8** (1998), 71–102.

[25] T. Peachey, D. Abramson and A. Lewis, Model optimization and parameter estimation with Nimrod/O, in: *International Conference on Computational Science*, University of Reading, UK, May 28–31, 2006.

[26] T.C. Peachey, N.T. Diamond, D. Abramson and C.M. Enticott, Computing factorial designs for large numbers of factors with 2^m levels, submitted for publication.

[27] P. Saha and L.L.R. Williams, Gravitational lensing model degeneracies: Is steepness all-important?, *The Astrophysical Journal* **653** (2006), 936–941.

[28] T. Santner, B. Williams and W. Notz, *The Design and Analysis of Computer Experiments*, Springer-Verlag, New York, 2003.

[29] M.W. Schmidt, K.K. Baldridge, J.A. Boatz, S.T. Elbert, M.S. Gordon, J.H. Jensen, S. Koseki, N. Matsunaga, K.A. Nguyen, S. Su, T.L. Windus, M. Dupuis and J.A. Montgomery, General atomic and molecular electronic structure system, *J. Comput. Chem.* **14** (1993), 1347–1363; http://www.msg.ameslab.gov/GAMESS/, accessed September 20, 2007.

[30] T.R. Shannon, F. Wang, J. Puglisi, C. Weber and D.M. Bers, A mathematical treatment of integrated Ca dynamics within the ventricular myocyte, *Biophysical Journal* **87** (2004), 351–371.

[31] W. Sudholt, K.K. Baldridge, D. Abramson, C. Enticott and S. Garic, Applying grid computing to the parameter sweep of a group difference pseudopotential, in: *Computational Science – ICCS 2004: 4th International Conference*, Kraków, Poland, June 6–9, 2004, Lecture Notes in Computer Science, Vol. 3036, 2004, pp. 148–155.

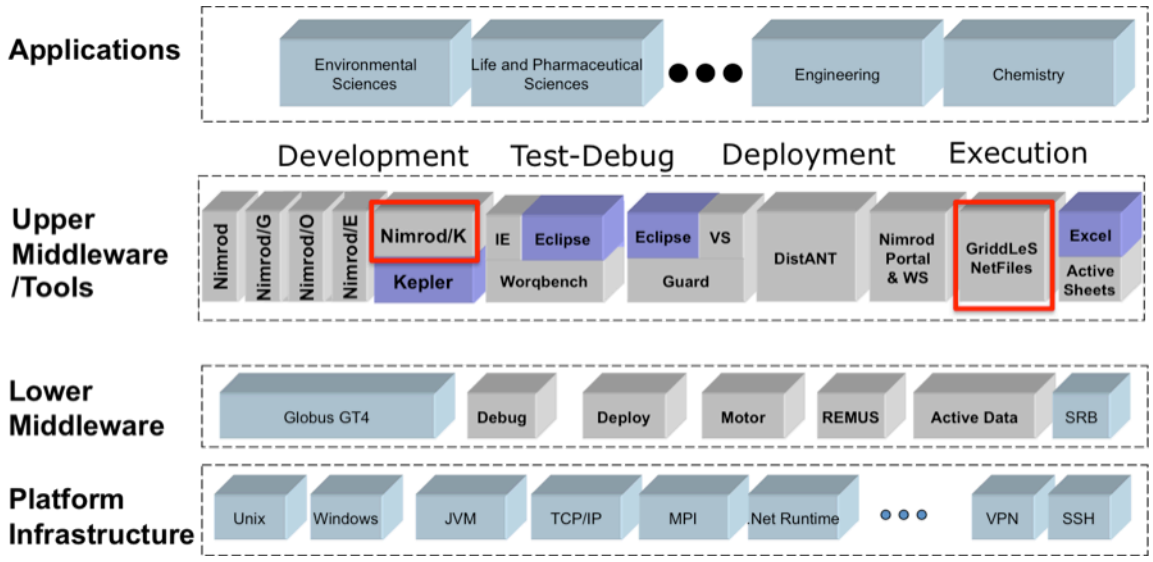
[32] W. Sudholt, K.K. Baldridge, D. Abramson, C. Enticott and S. Garic, Parameter scan of an effective group difference pseudopotential using grid computing, *New Generation Computing* **22** (2004), 125–136.

[33] W. Sudholt, K. Baldridge, D. Abramson, C. Enticott and S. Garic, Application of grid computing to parameter sweeps and optimizations in molecular modeling, *Future Generation Computer Systems* **21** (2005), 27–35.

[34] B. Vanderstraeten, P.W. Chin, M. Fix, A. Leal, G. Mora, N. Reynaert, J. Seco, M. Soukup, E. Spezi, W. De Neve and H. Thierens, Conversion of CT numbers into tissue parameters for Monte Carlo dose calculations: a multi-centre study, *Physics in Medicine and Biology* **52** (2007), 539–562.

[35] Y. Zhang, T.-S. Lee and W. Yang, A pseudobond approach to combining quantum mechanical and molecular mechanical methods, *J. Chem. Phys.* **110** (1999), 46–54.

Chapter 6: Nimrod/K and Scientific Workflows.



By the mid 2000s it was clear that scientific applications required a richer framework for describing experiments that originally designed into the existing Nimrod tools. Accordingly, I initiated work on adding parametric computing into scientific workflows. (21) describes the initial framework for doing this by supporting a range of flexible inter-component communication schemes. (22) discusses a scheduler that supports some workflow like semantics in the original tool chain, and (23) shows how I extended the execution machinery in an existing workflow tool, Kepler, to support parallel execution. (24) extends the framework to incorporate other parametric search methods supported in the Nimrod tools.

(1) Abramson, D. and Kommineni, J., "A Flexible IO Scheme for Grid Workflows". IPDPS-04, Santa Fe, New Mexico, April 2004. 6-1 .. 6-10

(2) Ayyub, S. and Abramson, D. "GridRod - A dynamic runtime scheduler for grid workflows". 21st ACM International Conference on Supercomputing, June 16-20, 2007, Seattle. 6-11 .. 6-20

(3) Abramson, D., Enticott, C and Altinas, I. "Nimrod/K: Towards Massively Parallel Dynamic Grid Workflows", IEEE Supercomputing 2008, Austin, Texas, November 2008. 6-21 .. 6-31

(4) Abramson, D., Bethwaite, B., Enticott, C., Garic, S and Peachey, T. "Parameter Space Exploration using Scientific workflows", ICCS 2009, SU Center for Computation & Technology, Baton Rouge, Louisiana, U.S.A, May 25-27, 2009. 6-32 .. 6-41

Contributions

Kommineni was a Masters student under my supervision; Ayyub was a PhD student under my supervision; Enticott, who was originally a research assistant, converted to study a PhD under my supervision; Altintas is a colleague from the Univ of California, San Diego, with expertise in workflows and assisted with the Kepler integration; Garic, Bethwaite, Dinh, Ramdas and Peachey were research associates under my supervision; Firth, Lackmann, Vail and Harper were colleagues from Monash who provided expertise in microscopy; Sheck was a colleague from Leica Micro-systems with expertise in microscopy; Michailova, Chitters and Amirriazi provided a case study in cardiac science.

A Flexible IO Scheme for Grid Workflows

David Abramson and Jagan Kommineni
School of Computer Science & Software Engineering,
Monash University, Clayton
Australia, 3800
{davida, jagan}@csse.monash.edu.au

Abstract

Computational Grids have been proposed as the next generation computing platform for solving large-scale problems in science, engineering, and commerce. There is an enormous amount of interest in applications, called Grid Workflows in which a number of otherwise independent programs are run in a "pipeline". In practice, there are a number of different mechanisms that can be used to couple the models, ranging from loosely coupled file based IO to tightly coupled message passing. In this paper we propose a flexible IO architecture that provides a wide range of mechanisms for building Grid Workflows without the need for any source code modification and without the need to fix them at design time. Further, the architecture works with legacy applications. We evaluate the performance of our prototype system using a workflow in computational mechanics.

1 Introduction

Computational and data Grids couple geographically distributed resources such as high performance computers, workstations, clusters, and scientific instruments. Accordingly, they have been proposed as the next generation computing platform for solving large-scale problems in science, engineering, and commerce [11][12] Unlike traditional high performance computing systems, such Grids provide more than just computing power, because they address issues of wide area networking, wide area scheduling and resource discovery in ways that allow many resources to be assembled on demand to solve large problems. Grid applications have the potential to allow real time processing of data streams from scientific instruments such as particle accelerators and telescopes in ways which are much more flexible and powerful that is currently available. A number of prototype applications have been built and these demonstrate that the Grid computing paradigm holds much promise [23].

There is an enormous amount of interest in applications, called Grid Workflows in which a number of otherwise independent programs are run in a "pipeline"; so much interest that this is currently being addressed by a working group with the Global Grid Forum [22][21]. Using this approach, it is possible to process data from an arbitrary source ranging from databases to real time data from scientific instruments. Workflows can be connected in a flexible and dynamic way to give the appearance of a *single application* spanning multiple physical organizations, with access to a wide range of data running on very high performance platforms. Grid workflows have been specified for a number of different scientific domains including physics [14], gravitational wave physics [9], geophysics [31], astronomy [3] and bioinformatics [29].

There is also a great deal of interest in reusing legacy applications within the workflow rather than rewriting new programs. Obviously, this is because we wish to leverage the enormous amount of complex scientific software that already exists. However, many of these existing programs have been written in a range of legacy languages such as Fortran and C and use conventional file IO operations like READ and WRITE. Further, many of them are quite old and are not well suited to modification. As a result, almost all of the Grid Workflow effort assumes that sub components will read and write local files, and that the data will be copied from one stage of the pipeline to the next, as is the case in Nimrod [1] and APST [6]. Whilst powerful, we are interested in supporting a wide range of IO mechanisms ranging from local files to message passing.

In this paper we propose a flexible IO architecture that provides a wide range of mechanisms for building Grid Workflows *without the need for any source code modification and without the need to fix the mechanisms at design time*. Our system is interposed between the application and conventional Grid middleware services like Globus, thus the components of the workflow believe they are executing in a conventional file system, but the system takes advantage of the flexibility and performance of the Grid.

In section 2 we discuss the type of IO operations we want to perform, and in section 3 we discuss the design of a module called a File Multiplexer (FM) that performs these. Section 4 discusses the implementation considerations, followed by an evaluation of the performance of the FM using two case studies.

This work is part of a large project called GriddLeS which aims to support the development of **Grid** applications using **L**egacy **S**oftware [17].

2 What sort of IO do we want to support?

Figure 1 shows hypothetical pipeline and illustrates the diversity in IO operations that may be required. Suppose we build a workflow that consists of three main computational phases, executed on three distinct computer systems. In this example, Phase 1 takes data from both a database, and a scientific instrument, processes it, and produces some output in a file. Phase 2 takes that output file as input, but also reads some input from one of a number of replicated data files. It chooses the copy that is as close as possible to Machine 2, where the work is being performed. Phase 2 writes some output files, some of which form the input to Phase 3, which is executed on another machine again.

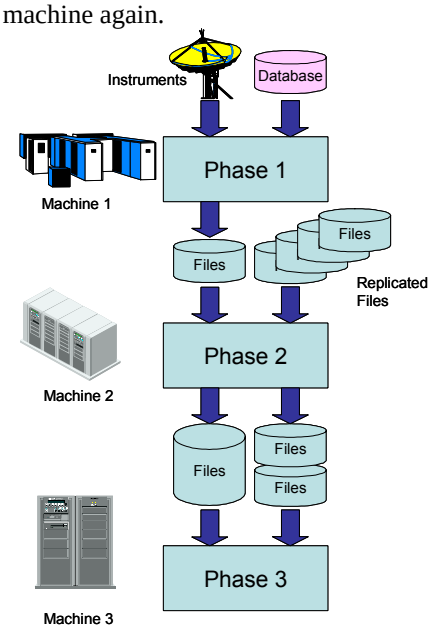


Figure 1 – Sample workflow

Clearly there are a number of ways that this pipeline can be implemented. Most likely, the Phase 1 code will need to be modified so that it can read data directly from the instrument. Likewise, Phase 1 will need code to access the database, which may be local or distributed onto other platforms. Since Phase 2 is executed on a different computer system to Phase 1, it is either necessary to copy files from Machine 1 to Machine 2, or there must be some sort of shared file system across these machines. Phase 2 also reads a file that is replicated on the Grid. In order to do this, it must first locate the most appropriate source, using some system like the Globus Replica Catalogue [32] or the Storage Resource Broker (SRB) [30], and then it must either copy the file to Machine 2 or use a remote access method like those provided in the SRB or Global Access to Secondary Storage (GASS) [19]. Further, in deciding which file to use, it might need to look at the dynamic bandwidth between systems using something like the Network Weather Service (NWS) [36]. Similarly, when Phase 3 executes on Machine 3, the output files from Phase 2 must be copied to the machines.

Alternatively, it may be beneficial to overlap the computations of Phases 1, 2 and 3 to form a genuine pipelined execution. To achieve this, we would need to remove the IO operations in the programs and replace them with something like those provided by MPICH-G2 [25].

This example is useful because it illustrates the variety of IO that we would like to be able to support. These can be summarized as:

1. Local file IO
2. Local file IO with files copied from one machine to another
3. Remote file IO
4. Remote replicated IO
5. Local replicated IO (i.e. find replica then copy)
6. Direct message passing

Moving between any of these different access methods would almost certainly require source code modification, something we are keen to avoid. Further, it may not be possible to decide which modes to use until the actual workflow is executed. For example, it may not be known whether data files are replicated, or how much bandwidth is available, until the programs are run. Likewise, the computations cannot be scheduled until the loads on various machines are known, so it may not be possible to decide whether a file is local or not before hand. Finally, deciding whether computations should be tightly coupled using message passing may depend on the the characteristics of the communication channels (bandwidth and latency) that are available between the machines that are chosen. Accordingly it is actually desirable to delay the choice of IO method until the application is configured and scheduled. Unfortunately, many of the IO modes discussed above have different APIs and access methods, and this would normally require source code modification. Notably, many of the workflow systems that are currently being discussed assume that files will be copied from one system to another, restricting the effective IO modes dramatically.

3 Flexible IO using the File Multiplexer

As discussed in the previous section we would like to be able to support a wide range of IO methods without modification of the source program. At one end of the spectrum, we want the program to perform local file operations as originally intended. At the other end of the spectrum we wish to connect readers and writers together with a direct TCP connection, without the need to write a file at all. Further, the decision about which mode to use needs to be delayed as late as possible, and may even be changed whilst the programs are running.

3.1 The File Multiplexer Architecture

The key to providing a flexible IO system is to interpose a library between the application and the Grid. We have called this library the File Multiplexer because it makes a dynamic choice about which methods to implement. Figure 2 shows the architecture of the FM.

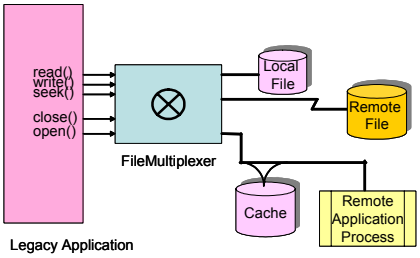


Figure 2 – The File Multiplexer (FM)

The FM intercepts all file operations as specified in the legacy application. When the program performs an OPEN operation, the FM determines which mode to use, and sets up the appropriate pathways. Each OPEN operation makes an independent choice, thus one file may be located locally and another may be remote.

If local file accesses are required, then all of the IO operations are simply diverted to the conventional IO library. Remote file operations can be supported in two ways. First, the remote file can be copied to the local machine, and then local operations can be performed. If the file is modified it can be copied back when it is CLOSED. Second, the FM can access the file on the remote machine using a proxy file server. This type of file operation is supported by systems like Condor. The choice of mode should be based on information about the access patterns and the file size. For example, if an application reads a small fraction of the remote file, it may not warrant copying it to the local file system. Further, if the file is very large, it may not be possible to copy it, and remote accesses might be more efficient. On the other hand, if a file is small and the latency to the remote system it high, then it is more efficient to copy the file, especially if a latency hiding method like GridFTP is available [19].

If a remote file is replicated, the FM needs to decide which one to access. A range of heuristics can be used to make this decision. For example, if dynamic information such as the network bandwidth and latency is available, then the most *efficient* pathway can be chosen. Further, if a file is opened in read-only mode, then the FM can actually change the mapping dynamically during the execution, allowing it to adapt to changing network conditions. A range of systems can provide replica information, such the Globus Replica Catalogue and the Storage Resource Broker. As with other remote IO, the FM has the choice of copying the file to the local file system or using a remote proxy server.

In some cases a downstream reader application may be overlapped with the writer. For example, in many scientific computations, data is written for each time step, and this is used immediately by a down stream computation. In this scenario, it is desirable to connect the writer directly to the downstream reader using a socket or message buffer. As blocks are written they are copied to the remote system and stored in a buffer. As they are read they can be removed from the buffer. Whilst it is actually possible to place the buffer at either the writer end or the reader end, it is usually more efficient to place it at the reader end, and by using an asynchronous write operation the latency between the systems can be masked. On the other hand, if a writer is broadcasting to a number of readers, such as when a file is used by more than one down stream process, it may be necessary to place the buffer at the reader end of the channel. Normally, such direct communication only works when both the writer and the reader perform sequential IO. However, by caching the file buffer with a cache file, it is possible to allow the reader to re-read blocks from the input stream, and even perform arbitrary seeks. Synchronization of such operations requires the creation and maintenance of a structure that records which blocks have been written, and whether a copy resided in the cache file or not. If a block has not been written, the reader must wait for it to arrive. Accordingly, the read operation can be blocked until the data is written. Similarly, if a reader wishes to seek back to the start of a file and re-read records, it can fetch these the second time around from the cache file. Figure 3 summarizes the use of direct connections.

Interestingly each of the methods discussed in this section have different performance characteristics, and support different concurrency models. Because the FM makes the decision about which method to use at run time, the overall performance of the system can be optimized.

3.2 Configuring a Workflow

When a file is opened, the FM needs to decide which access mode to use. This decision is based on two sorts of dynamic information. First, the designer needs to indicate the way they want the workflow configured. Specifically, they need to decide whether files are accessed locally or remotely, whether a file is replicated, and whether direct buffer connections are required. Most of this information needs to be specified after the components have been written and tested, but before the system is executed. Second, the system needs to obtain information like the machine loads and network performance statistics, and use these to tune where files are located. This information is highly dynamic and may change during the execution of the application.

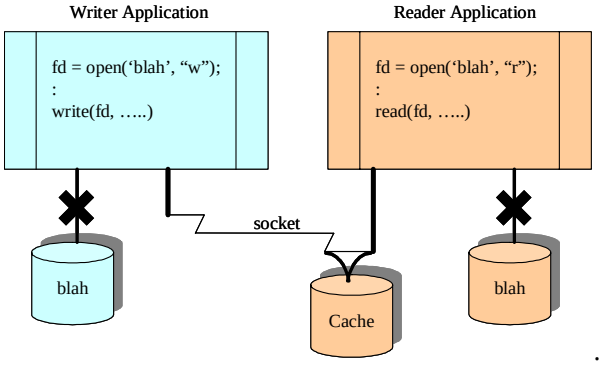


Figure 3 – Direct socket connections

Configuration information is held in a special database, which is accessed using a server call the GriddLeS Name Service (GNS). Each workflow may have its own GNS, or a single GNS may hold configuration information for a number of workflows. When an OPEN call is executed, the GNS is consulted. The GNS matches the names of the machine on which the code resides, and the full path name of the file in the OPEN call, and returns information to the FM about how to configure the IO. If local IO is required, it returns the name of the local file and instructs the FM to use the local file library. If remote access is required, information about the location of the remote file is returned. Further, remote replica information can be resolved at this time.

When a direct connection is requested, the system needs to connect the writer process to the corresponding reader process. To solve this problem we have developed a global naming scheme and have built a manager that recognises when writers and readers are referring to the same information. Once matched, the system returns the identity and location of the buffer.

3.3 Handling Heterogeneity

Because the Grid consists of machines with differing architectures and operating systems, it is important that the File Multiplexer operates in a heterogeneous environment. The most significant issue is the different byte ordering which has long since been a problem when binary data is taken from one machine to another. Traditionally, applications developers have solved this problem by using formatted ASCII data files when data needs to be written and read on different systems. However, it is possible to use unformatted binary data, and if the record structure is known the bytes can be reordered either before or after they are moved from one machine to another. Currently the File Multiplexer handles formatted ASCII data and binary data only if the two end points have the same byte ordering. However, we are experimenting with a scheme for describing the record structure so that the FM can reorder the bytes dynamically. The data would then be mapped into a neutral form as is done in XDR [18].

3.4 Related Work

A number of projects share features in common with GriddLeS and in particular the File Multiplexer. Intercepting file operations like this is not new, and many systems have used the same method to provide more flexible IO. The Legion file system copies files in before an application is run and out afterwards [35], as does Nimro/G [1]. On the other hand, Condor provides such a mechanism for accessing files in the home file system even when an application is run on a remote node [33]. Condor uses a library called BYPASS, which traps all file system traffic and directs it to a remote shadow process. BYPASS has been used in a number of projects, including The Pluggable File System (now known as Parrot) [16]. Many remote file systems have been built over the years, including NFS, AFS and more recently Grid file systems like the Globus GASS [20]. Many database systems support distributed access in which clients can address one or more database servers. Similarly, the Storage Resource Broker provides a variety of APIs for accessing archived replicated data transparently. All of these systems make it possible to access data from a number of machines, although the level of synchronization support varies between them.

There is much work on developing interoperability among high-performance scientific components by defining an interface definition language that supports scientific abstractions [13][4][15]. However, these techniques still require significant source code

modification, and are better suited to developing new codes rather than legacy file oriented applications.

None of the other work discussed here are as flexible as the approach being proposed in this paper. We are targeting six different IO mechanisms which can be reconfigured dynamically without the need to alter program source code.

4 Implementation Issues

Figure 4 provides an architectural overview of the implementation of the File Multiplexer and associated servers. Each application is linked to its own File Multiplexer library, which contains a number of client modules. As discussed, the normal file IO primitives are intercepted by the File Multiplexer, and these are processed either by the Local File Client, the Remote File Client or the Grid Buffer Client depending on whether the file reference is for a local file, a remote file or an inter-process TCP connection (accordingly).

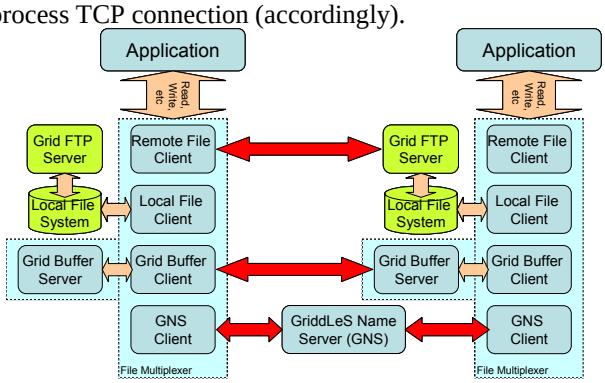


Figure 4 – Architecture of GriddLeS

The GNS Client is responsible for resolving the local file names specified in the OPEN calls, and for mapping these to either local files, remote files, remote replicated files or remote processes. The File Multiplexer treats the GNS as a read only database, and matches up multiple OPEN calls.

The Local File Client simply passes the calls onto the local file system, using the file name as resolved by the GNS. The Remote File Client connects to a Grid FTP server on the remote machine, and passes back blocks of the file as required. Note that the GridFTP server is a standard part of the Globus distribution, not a special component of GriddLeS. The Grid Buffer Client is responsible for implementing inter-process communication. It connects to a corresponding Grid Buffer Server on the other host, and sends blocks of data for each local WRITE call. At the other end of the socket, the Grid Buffer Client reads blocks by making calls to the local Grid Buffer Server. A cache file can be stored at either the sending end of a Grid Buffer connection or the receiving end. This provides considerable flexibility when faults occur. The buffer management code is able to support sequential read and write operations, but also if the cache is enabled, it can support out of order operations like SEEK. It also supports broadcast operations, where one application may write to the buffer, but many may read the buffer.

The GridBuffer service acts as a sink for WRITE operations and a source for READs. In order to support random read and write operations, data is stored in a hash table rather than a sequential buffer. The Grid Buffer Service is actually implemented using Web Services, and is accessed by SOAP messages. This implementation leverages the enormous effort in Web Services and also resolves difficulties experienced when direct TCP connections cannot be routed through firewalls.

The Bypass library discussed in section 3.4 provides a framework for intercepting the file system calls and also for accessing data at a remote site. We only utilized the code responsible for intercepting IO operations, and did not use any of the remote server code in our implementation. This is because we wanted to support more services than provided by Bypass.

5 A Case Studies

In this section we consider two computationally intensive workflows and illustrate the benefits of the File Multiplexer.

5.1 The Testbed

In these case studies we utilised a number of machines shown in Table 1, in four countries (AU, US, JP and UK). The machines consisted of different speed Intel Linux systems, connected by varying network speeds.

5.2 Mechanical Engineering

This case study considers computer models of thin plates containing holes and subject to cyclical loading. The models assume pre-existing cracks normal to the hole profile and use the Jones method of crack dynamics to estimate the number of cycles required for these cracks to spread from an initial length to some final length [24]. Our aim is to determine the hole shapes that will maximize the life of the worst (least cycles) crack. Previous work has shown that optimizing for life in this way may give different results from optimizing for stress on the hole boundary [7]. Figure 6 shows the stress distribution in the plate for a particular hole shape.

Name	Address & Details	Country
dione	dione.csse.monash.edu.au Pentium 4, 1500 MHz, 256 MB, Redhat Linux 7.3	AU
freak	freak.ucsd.edu Athlon Processor, 700 MHz, 256 MB, i386, Debian	US
vpac27	vpac27.vpac.org Pentium 3, 997 MHz, 256 MB, Red Hat Linux 7.3	AU
brecca	brecca-2.vpac.org Intel Xeon, 2.8 GHz, 2048MB, Redhat Linux 7.3	AU
bouscat	Bouscat.cs.cf.ac.uk Pentium 3, 1544 MB, 1 GHz, Red Hat Linux 7.2	UK
jagan	jagan.csse.monash.edu.au Pentium 3, 350 MHz 128 MB, Redhat Linux 7.3	AU
koume00	koume00.hpcc.jp Pentium 3, 1024 MB, 1400MHz Red Hat Linux 7.3, i686	JP

Table 1 – Machine List

In order to complete the computations, we need to execute a pipeline of 5 programs, as shown in Figure 5. CHAMMY takes a formula for a hole shape, depending on several parameters and generates points on the boundary of that hole. The programs MAKES_SF_FILES and OBJECTIVE are used to transform data from one phase to the other. PAFEC is a finite element code that computes the stress tensors in the meshed design. FAST is a crack propagation code that computes the number of cycles before a number of independently placed cracks reach a certain length.

Traditionally, the entire pipeline has been executed on the one computer, with intermediate results passed using files. Importantly, some files are passed from one phase to another, whereas, other files are simply read from the file system. The final output, RESULT.DAT, contains the value for the life of the design, which is the minimum time for any of the cracks to reach a certain length. This result defines the life of the design. The File Multiplexer and GNS are flexible enough to map some files to the local file system, whilst linking writer-reader file chains into direct TCP connections.

We performed a number of experiments as shown in Table 2. We recorded the times that each component completed, and the total execution time for the application

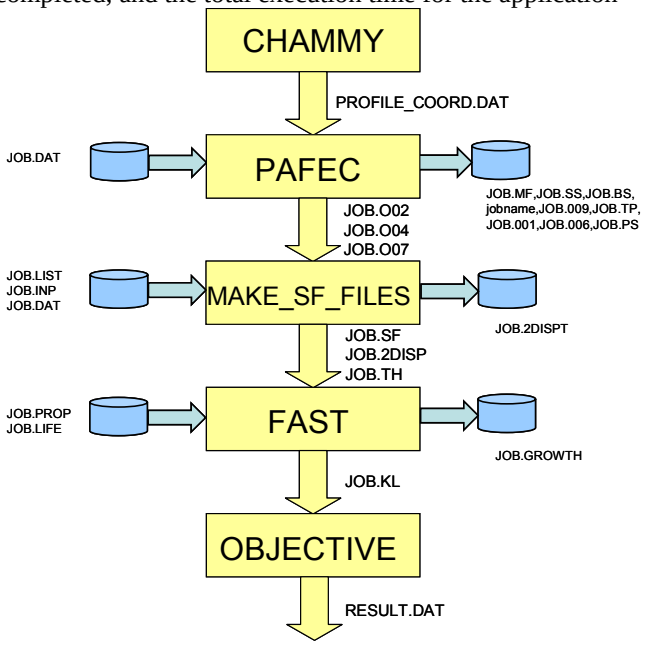


Figure 5 – Durability Pipeline

Exp	Assignment of tasks	Nature of IPC	Total Time mm:ss
1	All programs on jagan…	Files	99:17
2	All Programs on jagan…	GridFiles	89:17
3	Chammy on koume00… Pafrun on jagan… Make_sf_file on dione… Fast on vpac27… Objective on freak…	GridFiles	55:11

Table 2 – details of the computational experiments performed

In experiment 1 all phases of the computation are run on the machine “jagan”, at Monash University, Melbourne Australia. In this case, GriddLeS was configured so that local files were written and read by each phase of the computation. Accordingly, the next phase in the pipeline cannot start until the previous one is complete. In experiment 2, the programs were again run on the one machine, “jagan”, but direct connections were used instead of real files. This time the experiment completed 10 minutes faster, mostly because the next phase of the computation could start before the previous one completed. Thus, the phases were overlapped into a

pipeline. In experiment 3 all phases were run on different machines, which were physically distributed across three different countries. This experiment completed some 44 minutes faster than experiment 1. The speedup can be attributed to both the overlap in computation provided by using buffers, and also the machines were of varying speeds. This latter point is a powerful argument in support of such a computational grid, because sometimes faster machines are available at a remote location.

5.3 Atmospheric Sciences

In this case study we consider a workflow consisting of three atmospheric models, a Global Climate Model called C-CAM, a linking model called cc2lam and a regional weather model called DARLAM [27][28][34]. Traditionally, C-CAM and DARLAM have been executed on the same computer system, and data is passed between then using conventional files. However, the computational Grid provides an ideal framework for executing these models on different machines that are physically distributed. Importantly, there are many models in use by the atmospheric sciences community, and the Grid provides an opportunity to build genuine multi-organizational models from components that are "owned" by different partners.

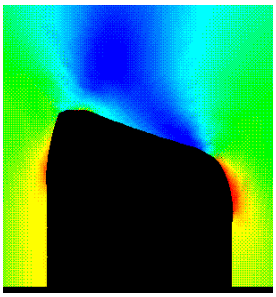


Figure 6 – Stress distribution for one particular hole shape

Importantly, most of the computation is performed by C-CAM and DARLAM, and cc2lam provides simple data manipulation and filtering between the two codes. In this section we describe an experiment in which C-CAM, cc2lam and DARLAM are coupled by Grid Buffers, that is, the output of C-CAM is streamed into DARLAM (via cc2lam). This configuration is shown in Figure 6. This is important because C-CAM and DARLAM are legacy codes written in Fortran, and we did not wish to make modifications to their structure.

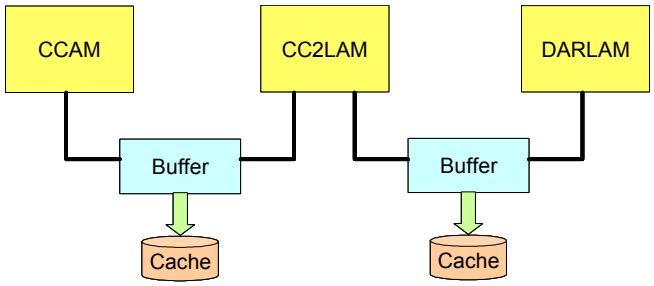


Figure 6 – Linking models using Grid Buffers

When the first writer application (C-CAM) writes a block of data (typically of 4096 bytes) using a write statement, the data is transferred to the Grid Buffer service using the client component of the Grid Buffer service. Once the data arrives, the reader application (in this context cc2lam) uses the Grid Buffer client to obtain the block of data and writes to the other Grid Buffer in a similar way. DARLAM behaves in a similar way, reading from the Grid Buffer and writing its output to conventional files. In some instances, DARLAM re-reads some of the input data. Because the data has already been deleted from the hash table in the Grid buffer Service, it is read form the cache file instead. This occurs transparently to the DARLAM model.

In this case study, three different experiments were performed.

Case 1: All the models are executed sequentially on dedicated machines and the comparative clock timings are as indicated in the following Table 3 below

Machine	C-CAM	cc2lam	DARLAM	Total
dione	00:28:21	00:00:08	00:13:16	00:41:45
brecca	00:16:34	00:00:08	00:07:46	00:24:24
freak	00:30:31	00:00:30	00:13:38	00:44:39
bouscat	01:07:29	00:00:12	00:31:52	01:39:33
vpac27	01:05:22	00:00:11	00:31:00	01:36:33

Table 3 – Times in hr:min:sec for sequential runs.

Case 2: All models are executed concurrently on the same machine. There are two variations on this experiment. First, the programs read and write conventional local files. The results of this experiment are shown in the "With Files" column in Table 4. Second, the programs use Grid Buffers instead of files. These times are shown in the Buffers column in Table 4. All times are cumulative, and thus the DARLAM time also indicates the total time taken.

The results shown in Table 4 highlight that using buffers is always faster than using files when the codes are run on

the same system. This is interesting because the models are multiprocessing the single CPU on these machines, and thus it suggests that buffers are allowing some overlap of computation and communication. Even more interesting is that most of the runs performed with buffers were actually faster than running the codes sequentially on the same platform, as shown in Table 1, again because of the overlap in IO and computation. The exceptions to this are those run on dione and vpac27, which are presumably because of the relative speed of the computation and the IO on these two machines.

Case 3: C-CAM and DARLAM are executed on different computers at different locations, whilst cc2lam is run on the same machine as C-CAM. Because there are a large number of potential pairings of machines, we have selected a few interesting ones and present the timing results in Table 5. Again all times are cumulative, and thus the DARLAM time also indicates the total time taken. In the case where local files are written we have also included the time taken to copy the files in the cumulative totals.

Machine	Model	Files	Buffers
dione	C-CAM	00:41:18	00:44:10
	cc2lam	00:41:56	00:44:15
	DARLAM	01:08:17	00:49:12
brecca	C-CAM	00:18:13	00:20:05
	cc2lam	00:18:25	00:20:12
	DARLAM	00:27:58	00:22:57
freak	C-CAM	00:34:35	00:35:21
	cc2lam	00:35:26	00:35:33
	DARLAM	00:52:39	00:40:30
bouscat	C-CAM	01:10:22	01:17:51
	cc2lam	01:10:39	01:18:10
	DARLAM	01:55:27	01:29:59
vpac27	C-CAM	01:39:28	01:51:11
	cc2lam	01:40:24	01:52:05
	DARLAM	02:44:49	02:15:15

Table 4 – Cumulative concurrent runs on the same system (time in hr:min:sec)

The results in Table 5 show that under some circumstances it is faster to run the codes sequentially and copy the files between phases of the computation. However, under other circumstances it is faster to couple the codes using buffers. Specifically, when the network connections are fast, and the latency relatively low, it is usually faster to couple the codes using buffers. However, when the network connections have a high latency, for example from Australia to the UK or US, it is actually faster to run the codes sequentially. We believe this is because the file copy sends larger blocks of data, and thus the performance is less sensitive to network latency. *This case study highlights the main advantage of our approach, namely, we are able to shift from using file copies to buffers by only changing some parameters in the GNS.* Both experiments used exactly the same source code. At the same time, we are investigating whether we can produce a version of the buffer code that is less sensitive to network latency.

5.4 Summary

Both of the case studies have illustrated the ability of the File Multiplexer to dynamically assign file IO to local files or direct connections between processes. They did not demonstrate the use of remote files, however, this has been shown in another paper [26]. *The most important result of these experiments is that the changes in configuration required no modification of the software application, and this was achieved solely by changing entries in the GNS.*

Machine	Model	Files	Buffers
dione	C-CAM	00:28:21	00:34:20
dione	cc2lam	00:28:29	00:34:32
	File Copy	00:29:19	
vpac27	DARLAM	01:00:29	00:48:47
brecca	C-CAM	00:16:34	00:18:05
brecca	cc2lam	00:16:42	00:18:12
	File Copy	00:17:32	
dione	DARLAM	00:30:48	00:25:10
brecca	C-CAM	00:16:34	00:20:51
brecca	cc2lam	00:16:42	01:05:17
	File Copy	00:24:12	
bouscat	DARLAM	00:56:04	01:10:21
dione	C-CAM	00:28:21	00:35:24
dione	cc2lam	00:28:29	00:35:30
	File Copy	00:29:19	
brecca	DARLAM	00:37:05	00:39:24
brecca	C-CAM	00:16:34	00:18:37
brecca	cc2lam	00:16:42	00:18:44
	File Copy	00:16:57	
vpac27	DARLAM	00:47:57	00:40:43
brecca	C-CAM	00:16:34	00:18:19
brecca	cc2lam	00:16:42	00:33:49
	File Copy	00:20:17	
freak	DARLAM	00:33:55	00:41:45

Table 5 – Cumulative concurrent runs on different systems

6 Conclusion

This paper has discussed one aspect of the GriddleS system, namely the way that it supports the linkage of a number of separate software components into single distributed Grid applications. We described the design and implementation of a File Multiplexer, a module that maps file system primitives into either local or remote files, or allows communication with remote processes. We demonstrated the effectiveness of the system using two computationally intensive applications.S

GriddLeS makes extensive use of existing middleware layers like Globus to provide the low level services it needs. Thus, it should not be viewed as a replacement for Grid middleware, but rather as a layer that interfaces between legacy code and such middleware. There are many other aspects of GriddLeS that have not been discussed in this paper. For example, tools are required for specifying and composing a new Grid application. We have not discussed scheduling of components to resources and assignment of data sources. Clearly, there has been much work in this area, and we should be able to leverage existing schedulers like the Condor DAGman [8]. Importantly, the scheduler needs to take account of whether the workflow is configured to copy files or use direct connections, since both impose different scheduling constraints. For example, if file copies are performed the computations need to be run sequentially. On the other hand, if buffers are used then they need to run at the same time.

We have not yet implemented the replication management within the FM. We plan to make use of either the Globus Replication catalogue or the Storage Resource Broker (SRB) to support data replication, and tools like the Network Weather Service to provide up to date information about the relative costs of accessing data. We plan to extend our earlier Nimrod/G work which uses an experimental computational economy to provide user driven quality of service goals [5]. Finally, we will develop a set of real Grid applications that utilize many resources, and distributed data and software components. Clearly the small case studies chosen in this paper are only the beginning.

Acknowledgements

The authors would like to acknowledge the work of their colleagues, in particular Mr Tom Peachey and Professor Rhys Jones who provided the mechanical engineering case study, and John L. McGregor and Jack Katzfey who provided the Atmospheric Sciences case study. This work is supported by Australian Research Council and Hewlett Packard under and ARC Linkage grant.

References

[1] Abramson, D., Giddy, J. and Kotler, L. “High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?”, International Parallel and Distributed Processing Symposium (IPDPS), pp 520-528, Cancun, Mexico, May 2000.

[2] Allcock, W., Bester, J., Bresnahan, J., Chervenak, A., Liming, L., Meder, S. and Tuecke, S. “GridFTP Protocol Specification. “, GGF GridFTP Working Group Document, September 2002. URL

[3] Annis, J., Zhao,Y. et al., "Applying Chimera Virtual Data Concepts to Cluster Finding in the Sloan Sky Survey," Technical Report GriPhyN-2002-05, 2002.

[4] Armstrong, R., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S. and Smolinski, B. “Toward a Common Component Architecture for High-Performance Scientific Computing”. 8th IEEE International Symposium on High-Performance Distributed Computing, Redondo Beach, CA, August 3-6, 1999.

[5] Buyya, R, Abramson, D., and Giddy, J., An Economy Driven Resource Management Architecture for Global Computational Power Grids, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, Nevada, USA, June 26 - 29, 2000.

[6] Casanova, H., Obertelli,G., Berman, F. and Wolski, R.”The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid”, Proceedings of the Super Computing Conference (SC'2000).

[7] Chaperon, P, Jones, R., Heller, M., Pitt, S. and Rose, F., “A methodology for structural optimisation with damage tolerance constraints”, Engineering Failure Analysis, 7, pp 281-300, 2000.

[8] Condor DAGman, http://www.cs.wisc.edu/condor/dagman/

[9] Deelman, E., Blackburn, K. et al., "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists," 11th Intl Symposium on High Performance Distributed Computing, 2002.

[10] Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Lazzarini, A., Arbree, A., Cavanaugh, R. and Koranda, S. “Mapping Abstract Complex Workflows onto Grid Environments”, In Journal of Grid Computing (to appear).

[11] Foster I. and Kesselman C., Globus: A Metacomputing Infrastructure Toolkit, International Journal of Supercomputer Applications, 11(2): 115-128, 1997.

[12] Foster, I., and Kesselman, C. (editors), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, USA, 1999.

[13] Gannon, D., Bramley, R., Stuckey, T.,Villacis, J. Balasubramanian, J., Akman, E., Breg, F., Diwan, S. and Govindaraju, M. "Component Architectures for Distributed Scientific Problem Solving," IEEE Computational Science and Engineering," 5(2): 50-63, 19998

[14] GriPhyN, www.griphyn.org

[15] http://www.cca-forum.org/

[16] http://www.cs.wisc.edu/condor/pfs

[17] http://www.csse.monash.edu.au/~davida/griddles

[18] http://www.faqs.org/rfcs/rfc1014.html

[19] http://www.globus.org/datagrid/gridftp.html

[20] http://www.globus.org/gass/

[21] http://www.isi.edu/~deelman/wfm-rg/wfm_charter.pdf

[22] http://www/ggf.org

[23] http://www-fp.mcs.anl.gov/sc2000_netchallenge/entries.htm

[24] Jones, R. and Peng, D. "A simple method for computing the stress intensity factors for cracks at notches", Engineering Failure Analysis, 9 (2002) 683-702.

[25] Karonis, N., Toonen, B. and Foster, I."MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface", Journal of Parallel and Distributed Computing (JPDC), Vol. 63, No. 5, pp. 551-563, May 2003..

[26] Kommineni, J. "Grid Applications: A Seamless Approach with Web Services, The APAC Conference on Advanced Computing, Grid Applications and eResearch, Royal Pines Resort, Gold Coast, Queensland Australia. 29 September - 2 October, 2003.

[27] McGregor, J. L., Nguyen, K. C., and Katzfey, J. J. Regional climate simulations using a stretched-grid global model. In: Research activities in atmospheric and oceanic modelling. H. Ritchie (ed.). (CAS/JSC Working Group on Numerical Experimentation Report; 32; WMO/TD - no. 1105) [Geneva]: WMO. p. 3.15-3.16, 2002.

[28] McGregor, J.L., Walsh, K.J. and Katzfey, J.J. Nested modelling for regional climate studies. In: A.J. Jakeman, M.B. Beck and M.J. McAleer (eds.), Modelling Change in Environmental Systems, J. Wiley and Sons, 367–386, 1993.

[29] NPACI, "Telescience, https://gridport.npaci.edu/Telescience/."

[30] Rajasekar, A., Wan, M., Moore, R., Kremenek, G. and Guptill,T. "Data Grids, Collections and Grid Bricks", 20th IEEE/ 11th NASA Goddard Conference on Mass Storage Systems & Technologies (MSST2003) San Diego, California, April 7-10, 2003.

[31] Southern California Earthquake Center's Community Modeling. Environment, "http://www.scec.org/cme/."

[32] Stockinger, H., Samar, A., Allcock, B., Foster, I., Holtman, K and Tierney, B, "File and Object Replication in Data Grids", Journal of Cluster Computing, 5(3)305-314, 2002.

[33] Thain, D and Livny, M. "Bypass: A tool for building split execution systems", In the Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing, Pittsburg, Pennsylvania, pp 79-85, August 1-4, 2000.

[34] Walsh, K.J. and McGregor, J.L. January and July climate simulations over the Australian region using a limited-area model. J. Climate, 8 (10), 2387–2403, 1995.

[35] White, B., Grimshaw, A., Nguyen-Tuong, A. "Grid-Based File Access: The Legion I/O Model", Ninth IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, Pennsylvania, Aug 1-4, 2000.

[36] Wolski, R., Spring, N. and Hayes, J., "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing", Journal of Future Generation Computing Systems,Volume 15, Numbers 5-6, pp. 757-768, October, 1999.

GridRod – A Dynamic Runtime Scheduler for Grid Workflows

Shahaan Ayyub
Monash University
MESSAGE LAB, Faculty of Information Technology,
Monash University Australia, 3145
+61 (03) 9903 1217
ayyub.shahaan@infotech.monash.edu.au

David Abramson
Monash University
MESSAGE LAB, Faculty of Information Technology,
Monash University Australia, 3145
+61 (03) 9905 1183
davida@infotech.monash.edu.au

ABSTRACT
Grid Workflows are emerging as practical programming models for solving large e-scientific problems on the Grid. However, it is typically assumed that the workflow components either read or write data to conventional files, which are copied from one execution stage to another, or they are tightly coupled using IPC libraries such as MPI or distributed streaming. More flexible communication can be achieved by overloading conventional READ and WRITE operations with advanced IO mechanisms such as sockets, streams and pipes, as is done in the GriddLeS environment. Such flexibility allows the pipelining of temporally dependent components, or in contrast, delaying of tightly coupled computations based on the current resource availability and network connectivity. However, it is also harder to schedule the workflow, because the communication mode may not be decided until run time. In this paper, we propose a new scheduling model that leverages such communication flexibility and allows us to generate dynamic runtime schedules. The scheduler in this case, not only allocates components to distributed Grid resources, but also specifies the inter-component communication mechanism (socket, pipe etc.) The current model is implemented as a dynamic workflow scheduling tool called **GridRod**, which harnesses Nimrod/G's [1] Grid services and GriddLeS [2] web services.

Categories and Subject Descriptors
I.2.8 [**Artificial Intelligence**]: Heuristic Methods and Scheduling – *runtime resource allocation and communication specification, spatio/temporal concurrency*.

General Terms
Algorithms, Management, Performance, Design, Reliability.

Keywords
Models of Computation, runtime scheduling, communication specification, spatial and temporal concurrency.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'07, Jun 18–20, 2007, Seattle, WA, USA.
Copyright (c) 2007 ACM 978-1-59593-768-1/07/0006 ...$5.00.

1. INTRODUCTION
The Grid Virtual Organization (VO) model integrates distributed resources such as high performance clusters, distributed data repositories, scientific instruments and specialized hardware devices to provide a collaborative platform to perform e-Science [3, 4]. Such platforms are used to solve large-scale problems by deploying complex application models, such as GENIE [1], specified as Grid workflows. The components of these workflows may consist of monolithic applications, data processing software, scientific instruments, visualization interfaces, or even remote Grid/Web services. Conceptually, formulation of this single "integrated application model" is fairly simple. However, its deployment and scheduling on highly distributed, dynamic and heterogeneous Grid environments is both interesting and challenging.

Accordingly, several projects such as ICENI [5], GridRPC [6] and VGrADS [2] implement a range of scheduling models for achieving high application performance whilst meeting such challenges. However, existing work does not allow the workflow to adapt to the underlying resource base that is available at execution time. Typically, there are two orthogonal approaches for modeling Grid workflows. The first approach temporally orders the workflow execution, where data is transferred from one execution stage to another as files. The second approach binds the components as co-executing computations communicating through unidirectional FIFO channels via pipes, streams or message passing. Both approaches tightly couple the communication and thereby restrict the runtime component allocation. Importantly, a designer specifies which approach to use statically when the workflow is built.

Communication libraries such as GriddLeS [2], allows the communication mode to be delayed until run time. This means that it is possible to choose the most appropriate mechanism depending on the resource base. For example, if there are sufficient resources, then it may be possible to co-schedule a number of the workflow components, and have these run concurrently. On the other hand, if there are not enough resources to run the components at the same time, then it is best to run each one sequentially, and to write results to intermediate files in between. GriddLeS provides this flexibility by intercepting and redirecting primitive IO operations (such as READ and WRITE) to a local file, a remote file or a remote socket. In this way, application components behave as if they are executing in a conventional file system whilst leveraging the distributed computational power of the Grid. Such flexible specification allows pipelining temporally dependent components or, in contrast, delaying tightly coupled co-executing computations. Furthermore, this exposes both temporal and spatial concurrency

in the Grid workflows, which can be exploited for achieving high throughput. Dynamic specification of inter-component communication in this way provides more opportunities to optimize runtime component allocation based on the current Grid state.

In this paper, we propose a new model, which makes flexible runtime decisions for scheduling Grid workflow components and for specifying inter-component communication behaviour. The model leverages the IO mechanisms already provided by GriddLeS to generate lazy runtime schedules. In a lazy approach the runtime scheduling decisions are delayed as much as possible. This allows the scheduling model to use the current information about the resource availability and network connectivity for achieving the optimisation. So, in cases where the data generation and consumption is continuous, the scheduler pipelines the distributed components when sufficient computational and network resources are available. In contrast, if the application components have producer-consumer relationships, then the model delays the execution of the downstream computation in case of resource unavailability or in order to optimise the execution overlap. So, if the data transfer on IO channels is unidirectional, the data can be temporarily stored in a buffer at the writer's end and later copied to the reader's location for its execution when sufficient resources become available. This would remove the necessity to execute the tightly coupled applications together, providing opportunities to optimise scheduling as well as improving resource utilization and consequently, applications performance. The scheduling function in this case is therefore a composite function, which not only allocates computational components to Grid resources, but also specifies how they should interact. The proposed model is embedded in the Nimrod/G framework [1], leveraging the flexible IO infrastructure already provided by GriddLeS. This infrastructure provides opportunities for spatial (parallelism) as well as temporal (pipelining) co-execution of the components. In order to maximize the throughput, both types of concurrency should be exploited. However, most Grid scheduling heuristics [2, 5, 7] [8, 9] tend to search in only one of the two orthogonal directions, exploiting only one type of concurrency. To address this, we propose a new class of scheduling heuristics called HyBD, which represents the hybrid of Breadth First and Depth First. The heuristics iteratively explores the workflow graph in both directions to optimise scheduling. Two novel heuristics, HyBD_MAKESPAN and HyBD_DELAY, are proposed under this class with different objective functions. The resulting infrastructure called GridRod is a step towards building a service-oriented architecture (SOA) for Grid Workflow Orchestration. GridRod leverages GriddLeS web services and Nimrod/G's Grid services to achieve the same.

2. Grid Workflows; An orchestration perspective

This section discusses Grid Workflows and workflow modeling techniques mainly from the orchestration and scheduling perspective. Synchronous Data Flow (SDF) [10] and Kahn process Nets (PN) [11], the two most prevalent workflow-modeling approaches, are described. These models are inherited from the set of existing models of computation in Digital Signal Processing (DSP). Variants of these models have been used in the projects such as Kepler [7] and ICENI for specifying Grid workflows, with some nomenclature differences. In the rest of the paper, we use the Kepler nomenclature (PN and SDF) as a reference.

General definition: We define Grid workflows as an integration of distributed standalone components interacting with each other by exchanging data through flexible communication links. The workflow is initially defined in an abstract form followed by the concrete mapping or triggering of components by an orchestration engine to appropriate Grid resources, whilst respecting dependencies. Also, Grid workflows can be represented as Directed Acyclic Graphs (DAGs). In this perspective, workflow modeling defines the workflow components as DAG nodes and their relationships as edges.

2.1 Grid Workflow Models

- *SDF* – Components in Grid-based SDF models are interconnected as temporally dependent units. Each component reads and writes data to a local file, in which case the inter-component communication is established by simply copying the data generated by an upstream writer to the reader's location. Currently, SDF is the most common model for Grid workflow specification as it allows integration of isolated components using a simple reader/writer relationship. However, for the very same reason, pipelining between these components becomes challenging.
- *PN* – Components in a PN model are linked by unidirectional first-in-first-out (FIFO) channels. All components communicate through these channels and their execution is synchronized (by essentially blocking and unblocking the reader process) depending on the data availability on the input channels. Examples of Grid based process networks include distributed streaming applications such as *count-Samps* [12] and *Clust-streams* [13]. These applications occur in scientific areas where a large amount of data is being continuously generated and must be processed in real-time. The Gates project [14], specifically targets scheduling and migration of these applications. However, it is assumed that applications modeled under PN have a communication layer already present between the components for coherent inter-component communication.

2.2 Static Modeling

Current Grid workflow modeling approaches tightly couple the inter-component communication, restricting the lazy runtime scheduling based on the latest CPU and network state. So, if the workflow is modeled as an SDF or a PN, then the components should communicate according to the tight coupling entailed by the model. This limits their dynamic allocation to distributed resources. A Grid workflow may well be constituted from a variety of components and therefore may have its sections modeled differently (as SDF or PN).

As mentioned earlier, if primitive I/Os such as read () and write () can be overridden by sockets, streams and pipes, then temporally dependent computations (modeled as SDF) can be pipelined provided the data is written and read continuously and sufficient resources are available. In this way SDF applications can behave as a network of dataflow processes, which is a special case of PN, and the main focus of our current work.

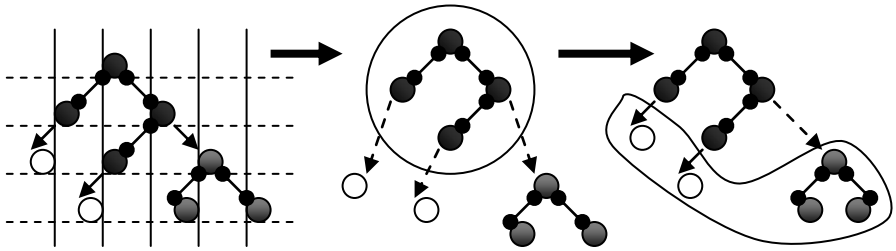


Figure 1. The graph has symmetric depth and breadth. The breadth of a graph defines the spatial concurrency between the components. The different shades of Gray represent two different clusters of PN within the same graph. The round-ended arrows mean that the two connected components can be streamed and concurrently executed. The large arrows show different scheduling states.

PN models, while providing concurrency and parallelism, tightly couple the communication between application components, which are assumed to interact using a pre-existing communication mechanism. Further, co-scheduling workflow components might not be optimal at all times. Consider a situation where a downstream reader takes less time to execute than its parent writer, or where the upstream writer takes only half of its execution period to generate the data that is to be consumed by the reader. Co-scheduling the components in these situations (a typical static co-scheduling approach [15]) would hold the reader's resource while waiting for the data availability, leading to poor resource utilization. Further, it is very likely, in highly dynamic environments, that by the time the reader is actually ready for execution, its resource is no longer the 'best resource' or even available for the execution. In these scenarios, a better approach would be to delay the execution of the downstream reader.

The knowledge of inherent communication behavior of the components is important in optimizing the scheduling. Also, such knowledge can be utilized to understand the bandwidth/latency and data requirements of the components thereby providing more chances for the scheduler to make optimized runtime decisions.

In the next section we describe our scheduling model, which dynamically schedules workflow components as clusters of PN and SDF based on their communication behavior. The scheduling model leverages the IO infrastructure provided by GriddLeS to achieve such modeling. Applications exhibiting asynchronous read/write patterns specifically benefit from such scheduling.

3. Dynamic Scheduling Model

Our model adopts a two-step process to optimize runtime scheduling of the workflow components. The first step is the clustering, which involves analytical selection of downstream components based on their communication behavior. The second step involves the following:

- Allocation of the selected components to Grid resources based on data availability and CPU performance.
- Selection of the appropriate communication mechanism to achieve better runtime application performance.

A new heuristic model called HyBD is proposed in this paper and two novel heuristics based on the proposed model are described.

3.1 HyBD Heuristics

HyBD stands for **Hy**brid of **B**readth First and **D**epth First, and signifies a hybrid of the two prevalent search approaches. A problem space can be described as a Directed Acyclic Graph (DAG) in at least two orthogonal representations, namely spatial and temporal [16]. In a spatial representation, all the components exist simultaneously and are executed concurrently, (spatial parallelism). On the other hand, in a temporal representation, the components have temporal precedence and are executed either in a sequential or a concurrent pipeline, depending on their communication behavior. The key to improve throughput is to explore both types of concurrency in order to optimize scheduling. However, existing graph exploration algorithms search in *either* of the two orthogonal directions (spatial or temporal), rather than in both. Thus an algorithm such as breadth first would explore the DAG spatially, whereas, the depth first would search it temporally. On the other hand, we are interested in finding a local optimization and therefore intend to explore the graph in both directions. For this, we have developed variants of Breadth and Depth First search approaches, which performs repeated searches along the orthogonal directions until a halting condition is encountered. In which case, each repetition essentially appends the visited nodes to a node-cluster list. However, the ordering in which the nodes are visited depends on the search method and the depth and breadth limits are obtained dynamically. This approach provides us with local search completeness to optimize component clustering.

3.2 Component Clustering

The clustering depends on the data generation and consumption patterns of the workflow components. Oldfield and Koltz [17], outline several scientific applications and their I/O behavior, including medical applications, seismic imaging, climate modeling, computational chemistry and biology. The report implies that most applications show asynchronous I/O patterns, and the basic read/write operations are *partially ordered*. Also, from our experiences in executing climate modeling applications [2, 18], we observed that some applications generate data continuously, whereas others perform this operation in single or discrete phases. This information is crucial and can be used for making clustering decisions in order to optimize the overlap between component executions.

Our scheduling model clusters the downstream components, which consume and generate data continuously as PN, and leaves the rest to behave as SDF (See Stages in Figure 1). The traversal is performed iteratively on every element of the set of temporally independent nodes. A child node becomes temporally independent once its parent finishes execution. The traversal stops if sufficient resources are not available or the heuristic decides to delay the execution of a downstream component due to data unavailability. This leaves the unscheduled downstream components waiting for resource or data availability. The clustered components are subsequently allocated and executed on distributed resources. At

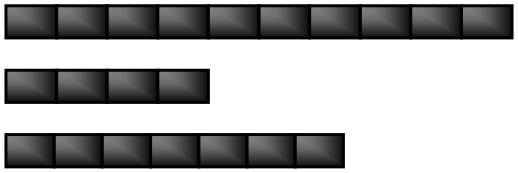


Figure 2 (a). Applications with different cost units. *Block ~ Cost unit.*

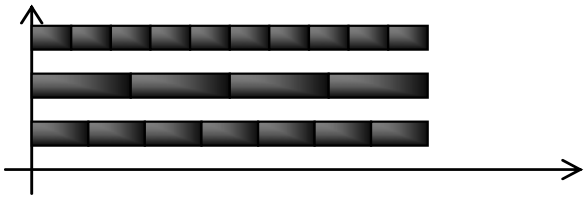


Figure 2(b). The Normalization Effect; The computation time of downstream computation is getting normalized. The total make-span is 10 cost units, however, with an additional wait time.

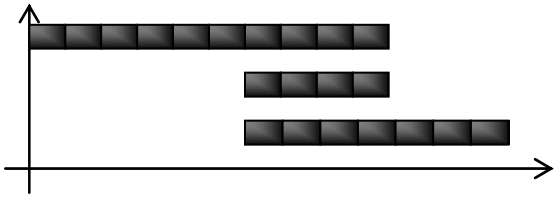


Figure 2(c). The Delay Effect. The delay in the downstream computation accrued an additional time of 3 cost units in the total make span. Nevertheless saving the CPU hours.

this point we propose that our scheduling approach is different from the conventional gang scheduling [19] as well as Gates' algorithm [14], as we assume that the components are non-pre-emptive applications and cannot be migrated once they start execution.

Ideally, if there are sufficient resources available, all clustered components can be co-scheduled; in which case, the execution time of downstream computations within the cluster would be normalized to the computation time of its parent. Figure 2(b) illustrates this scenario where jobs below are automatically stretched to the length of their respective parents. We would call this the *normalization effect* and would use it as a reference in the experiments section. This approach reduces the overall application make-span. However, it wastes CPU time because downstream computations have to wait for the data to arrive. Regardless, if no resources are available for further scheduling, then the delay is automatic. In situations where resource utilization is more important than reducing the application make-span, a delay may well be enforced explicitly. Thus, in cases where the estimated computation time of a downstream reader is lower than its parent, then it can be delayed for the difference in their execution time. This improves CPU utilization, at the expense of an increase in the application make span. We call this the *delay effect* as shown in Figure 2(c).

The third possibility is that a downstream computation is scheduled only after its parent finishes execution such as in SDF. This situation is likely to occur in cases where the upstream computation generates data in a single phase, which is generally at the end of its execution.

Depending on the possibilities described above, the application components are clustered and further allocated to distributed resources. However, in every case, the scheduling of downstream computations is delayed as much as possible by exploiting the components' inherent communication patterns. This delay is desired in order to get the latest system (network and CPU) information to generate better schedules. Grid workflows consisting of components with different write/read behaviors would specifically benefit from such an approach by optimizing the execution overlap between components, resulting in an improved workflow performance.

3.3 Component Allocation and Communication Specification

The second step involves allocation of the clustered components to best resources. This step also specifies how the scheduled components would communicate i.e. which GriddLeS IO mechanism should be implemented.

3.3.1 Component Allocation

Allocating components based on their data dependency can significantly improve application performance by reducing the communication overheads. In many scientific areas such as astronomy, geographic information systems and earth systems, data sets characterize the regions of the problem space. As a result, components processing common data sets exhibit spatial proximity. Luiz et al. [20] have demonstrated that allocating components based on such spatial proximity can reduce the application make-span. This is specifically true when the dependency between components in terms of file sharing is high. Casanova et al in [21, 22] demonstrate the efficiency of pre-staging and reusing the shared files to reduce data transfer overheads. We also base the allocation of clustered components on the similar criteria to achieve application performance as well as data and resource utilization. Once an optimum resource is chosen, then the communication behavior is dynamically specified based on current network and CPU states as described below.

3.3.2 Communication Specification

In the case of PN, there are several ways in which communication behavior can be specified. Depending on the location of the reader and writer, the IO can be redirected to a local file or remote file or a pipe. However, when remote communication is performed, a single point of network or CPU failure, which is very common in Grid environments, would require rescheduling of the components. This approach is clearly not very fault tolerant; would waste a lot of CPU time and incur additional data transfer costs. An alternative approach is to interpose a buffer at both the reader and writer's end. So, if the writing/reading is unidirectional then the writer and reader can uninterruptedly write and read data from their respective local buffers without getting affected by network failures. The proxy mechanism [23] implemented in Griddles allows such communication and also ensures data transfer and communication synchronization between the writer and reader buffers. Regardless of the time a downstream reader takes to get ready for the execution, the data which has been already generated on the writer's end can be copied using

```
Global Variables
Sn ← |V| {set of components or nodes}
R ← N {number of Resources}
Cp ← Cp is a subset of |E| {set of parallel edges}
Cs ← Cs is a subset of |E| {set of sequential edges}
procedure schedule
while Sn ≠ Ø d o
 Ssj← 0      {set of sorted ready jobs}
 Ssj← call(getReadyJobs,NULL) {get Sorted Ready Jobs}
 R ← R + call(relinquishResources,NULL)\n
 {relinquish resources from done jobs}
 SBF ← Ø {set of clustered jobs from BF traversal}
 CBF ← 0 {total cost units for Breadth First}
 SDF ← Ø {set of clustered jobs from DF traversal}
 CDF ← 0 {total cost units for Depth First}
 call(traverseBF, Ssj, CBF, SBF)

    for each pj ∈ Ssj do

    CSDF ← 0 {total cost units for
    Depth First for single seed}
    SSDF ← Ø {set of clustered
    jobs from DF traversal for single seed}
    call(traverseDF, pj,, CSDF, SSDF)
    SDF ←  SDF + SSDF
    CDF ←  CDF + CSDF
 end for

 if CBF < CDF then
     execute SDF
     Sn ←   Sn - SDF
 else
     execute SBF
     Sn ←   Sn - SBF
 end if
end while
*****************************************************************

procedure traverseDF(p, ., CDF, SDF)
 Jc ← {q : for all p→q, set of all children of p}
          for each q ∈ Jc:
     if CHILDREN(q) ≠ Ø do
         traverseDF(q,, CBF, SBF)
     else
         r ← allocate(q)
         if x do
    CDF += c(q) {c: cost unit}
    SDF ← SDF + q
    R ← R - r {remove chosen resource}
         end if
     end if
 end for
```

```
procedure TraverseBF (Ssj, CBF, SBF):
    LBF ← { Ssj } {LBF set of BF jobs; local variable}
    while LBF ≠ Ø do
        for each p ∈ LBF
            LBF ← LBF + CHILDREN(p)
            x ← allocate(p)
            if x do
                CBF += c(p) {c: cost unit}
                SBF ← SBF + p
                R ← R - r {remove chosen resource}
            end if
        end for
    end while

*****************************************************************

procedure Allocate (p)
    if !R do
        return 0
    end if
    q ← PARENT(p)    {q: parent of p}
    if (Heuristic == HyBD_DELAY)&& c(p) < c(q) do
        return 0
    else
        r ← call(chooseResource, p, R) {select resource for p
from R}
        call(chooseCommunication, p, q)\n
        {select appropriate communication between the p and q}
        return r
    end if

*****************************************************************

Procedure getReadyJobs (NULL)
    for each p in Sn do
        q ← PARENT(p)
        e ← p→q {e: the edge between p and q}
        if STATUS(q) in ('done','executing') do
            if e ∈ Cp && R > 1 && do
                STATUS(p) ← 'ready'
            else if e ∈ Cs && R > 1 && STATUS(q) = 'done' do
                STATUS(p) ← 'ready'
            end if
        else if STATUS(p) == 'pending' do
            if ! PENDINGTIME(p) do
                STATUS(p) ← 'ready'
        else
            STATUS(p) ← STATUS(p)

        end if
    end for
    return Sn
```

Figure 3. Pseudo Code

specialized multi-channel file transfer mechanisms such as GridFTP [24], also supported in GriddLeS, to the reader's buffer. The reader can therefore begin execution assuming that the file is available at its end. Similarly, in the case of SDF, data generated by the writer can also be copied using such specialized copy mechanisms.

Once the components are scheduled and communication between them is specified, appropriate Grid/Web services can be invoked to perform component execution and communication. Figure 1 shows temporal instances of the workflow DAG where parts of it are scheduled as PN and the rest of them are behaving as SDF.

3.4 Scheduling Heuristics

This section describes two new scheduling heuristics, namely HyBD_MAKESPAN and HyBD_DELAY, each having a different objective function for task allocation. These heuristics are prototypes of a class of heuristics, called HyBD, which we have described earlier in this paper.

3.4.1 HyBD_MAKESPAN

HyBD_MAKESPAN focuses on minimizing the total application make-span giving less priority to CPU utilization. Accordingly, it co-schedules as many components as possible without any delay. This results in CPU overheads as some of the downstream computations hold the resource whilst waiting for the data arrival at their respective input channels, owing to the normalization effect explained earlier. However, our simulation results show that the overall make-span of the application is reduced by this heuristic at the expense of additional CPU cost.

3.4.2 HyBD_DELAY

The primary objective of HyBD_DELAY is to maximize CPU utilization at the expense of additional delay in the application make-span. Nevertheless, this approach is beneficial in situations where the CPU costs are high, in which case the component allocation can be optimized according to its computational requirements.

The pseudo code of HyBD heuristics is shown in Figure 3.

4. Implementation; GridRod

The conceptual model described above has been implemented as a workflow orchestration tool called GridRod (see Figure 4). GridRod integrates and harnesses **GriddLeS** web services and Nim**rod**/G's Grid services to perform orchestration operations. Apart from high-level operations such as component allocation and communication specification, the integration allowed leveraging of already implemented low-level orchestration services in GriddLeS and Nimrod/G. These services include job launching, job execution and communication establishment. The proposed scheduling model has been embedded in Nimrod/G and the new heuristics were added to the already existing suit of heuristics to schedule workflow components.

After the components are allocated to suitable resources, the scheduling module considers the underlying resource infrastructure (e.g. Globus [1], Condor [25]) and invokes appropriate actuation services to launch the components (Figure 4, components x, y and z) on remote resources (Figure 4, edgeJL). Likewise, the scheduling module also calls the GriddLeS web service to interpose appropriate IO mechanisms (edgeCS) for inter-component communication. For example, edgeC4 in Figure 4 represents a local file mapping, in which case, comp y and z read and write data to a local file, and the execution coherence is handled by GriddLeS. Similarly, edgeC1 represents a direct socket, edgeC2 a direct file copy and edgeC3 a proxy-based data transfer mapping. Each communication edge (edgeCn where n Є {1,2,3,4}) is specified as a logical entry in the GriddLeS Naming Service (GNS) [2]. These entries are further used by GriddLeS to interpose the specified communication mechanism. GridRod utilizes Nimrod/G as the experiment launch pad and GriddLeS as its basic communication layer to handle inter-component interaction.

5. Related Work

In this section we describe some of the related work, which complements our proposed model. However, to the best of our literature review, we could not find a direct comparison with our model, and we therefore performed an empirical analysis.

5.1 Co-scheduling Overview

The DAG co-scheduling problem is not new, and there exists a number of static [9, 26] and dynamic [25, 27] models for scheduling Grid workflow components. Grid workflow components are primarily non-preemptive applications and the data arrival and consumption times on communication links are unpredictable. This makes their scheduling on dynamic and heterogeneous platforms such as the Grid NP-Hard. Consequently, most of the scheduling effort focuses on developing the "heuristics" that targets a near optimal solution to the problem. These heuristics can be categorized as follows

- List Based – HEFT [28]
- Clustering Based – Gang Scheduling [19]
- Implicit Co-scheduling heuristics – Spin Block [29]
- Partition Based – Pegasus [30]
- First In First Out Based – GridRPC [6]
- Ordering Based – ICENI [5], Condor-DagMAN [31]
- Data dependency based – APST [21]

List based algorithms generate static schedules, which are ineffective under dynamic network and resource conditions. There is a chance, for example, that the resource on which a component was statically scheduled is no longer optimal or available at the time of component execution. Clustering-based algorithms (e.g. gang scheduling [19]) involves scheduling of pre-emptive processes, however, we assume that the workflow components are non pre-emptive applications. Similarly, implicit co-scheduling schemes, implemented primarily in cluster environments, assume that the components interact with each other by passing messages, e.g., using MPI, and by blocking and unblocking their execution, something that does not apply to our situation.

Pegasus implements a just-in-time level-based task partitioning algorithm to schedule workflow components. However, arbitrarily partitioning the graph and clustering tasks based on levels delays the execution of the whole cluster just because of one task waiting for its execution trigger. On the other hand, the FIFO-based GridRPC co-scheduling model is very simple and does not consider issues such as network and computational resource performance when making scheduling decisions.

Also, all of these scheduling models, including ICENI and APST, are designed to optimise the task allocation and have limited control over the communication specification. As mentioned earlier, such static modelling restricts the scheduling of workflow components either as SDF or PN. However, we are interested not only in dynamically allocating the components but also in specifying how and when the components should communicate, which made a direct comparison of our model with other work difficult.

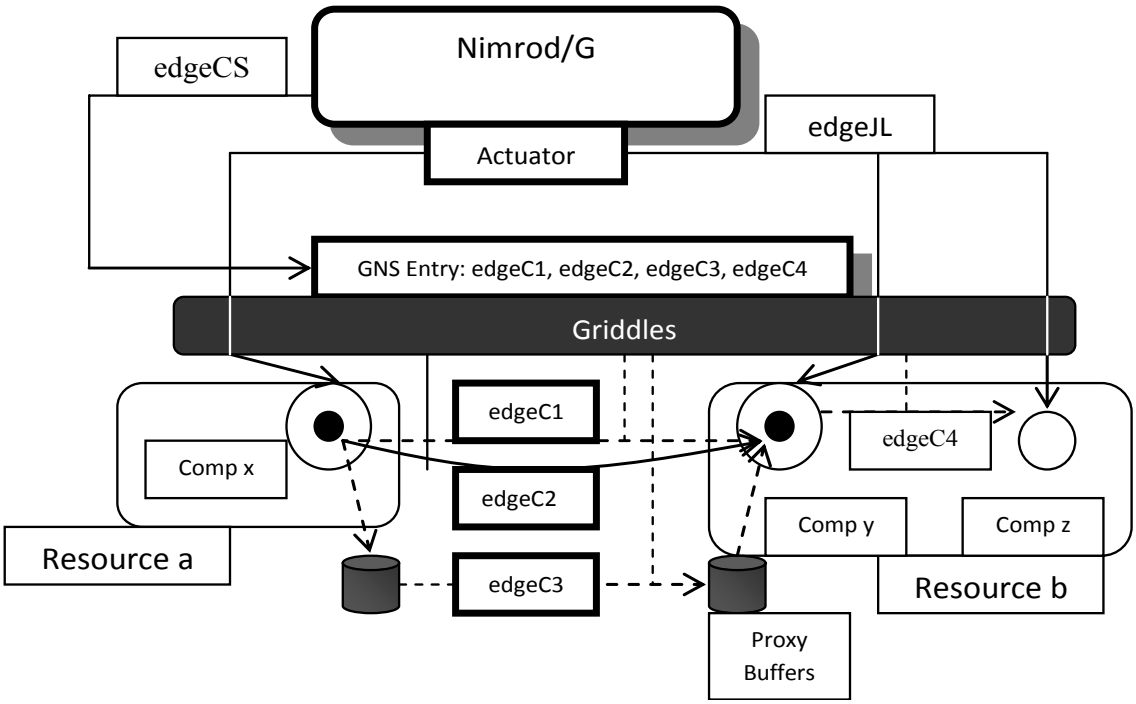


Figure 4. GridRod Architecture

An empirical experimental analysis of our scheduling model was performed. Non-iterative variants of simple Breadth First and Depth First algorithms were developed, which provided us with bounds to compare our heuristics. The non-iterative variants search the graph in one of the two orthogonal directions. However, they use the common task allocation and communication specification functions used by the HyBD heuristics.

6. Evaluation

6.1 Evaluation Metrics

The metrics against which we compared our proposed heuristics are described below:

1. Application make-span: We propose that, by exploiting the spatial and temporal parallelism between application components, the total make-span of the application can be significantly reduced. This makes application make-span a suitable metric to compare how well a heuristic explores the problem space and thereby improves the overall performance.
2. CPU utilization: We measured this metric in terms of cumulative execution time of all the components in the application. Both CPU utilization and cumulative execution time are inversely proportional to each other, which means that a high cumulative cost is equal to low CPU utilization.

6.2 Random Graph Generator

As described earlier, a workflow may have different concurrency patterns (temporal and spatial) between components, depending on their inherent communication behavior. Temporal concurrency refers to streamed pipelining whereas spatial concurrency represents parallelism. In order to generate graphs with varied concurrency patterns, we developed a random graph generator. The parameters for graph generation are the following:

6.2.1 Aspect Ratio

This parameter defines the ratio of breadth[1] vs. depth in the graph, and is specified as a value between 0 and 1. A value of 0 corresponds to a completely parallel application (breadth = number of nodes), e.g. a Parameter Sweep Application (PSA) (See figure 5(a) with breadth = number of nodes –1). On the other hand, a value of 1 corresponds to a sequential application (depth = number of nodes) (See figure 5(b)). A value of 0.5 generates the graph with a balanced depth vs. breadth ratio. The aspect ratio also determines the out-degree of the nodes, although we assume that all the nodes have unit or 1 in-degree. We are aware of the usage of the term aspect ratio in other literature such as graph drawing and image manipulation [32], and suggest that our definition is different from them. To the best of our literature search in graph generation, we could not find any terminology, which defines the aspect of breadth vs. depth, and therefore proposed our own definition.

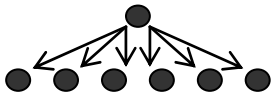


Figure 5(a)



Figure 5 (b)

[1] The Breadth describes the spatial parallelism of the graph.

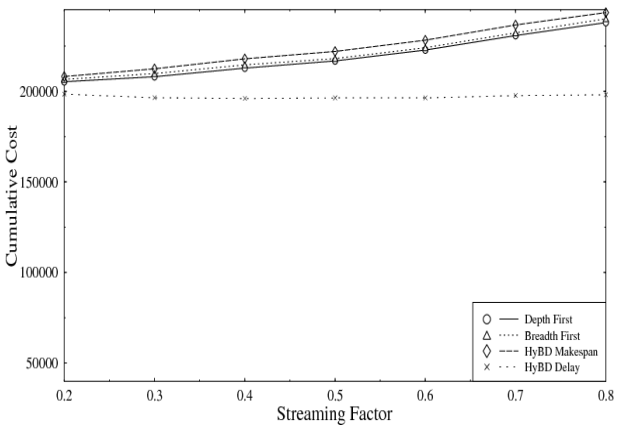


Figure 6(a). Cumulative cost vs. streaming factor

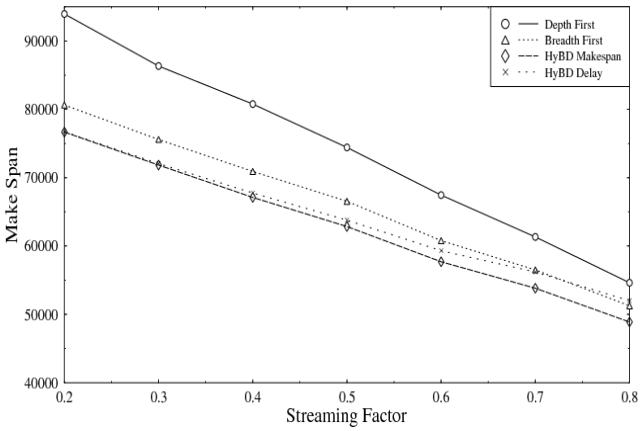


Figure 6(b). Make span vs. Streaming factor

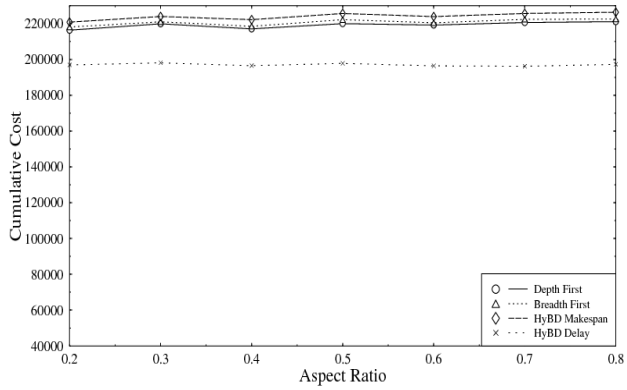


Figure 7(a). Cumulative cost vs. Aspect ratio

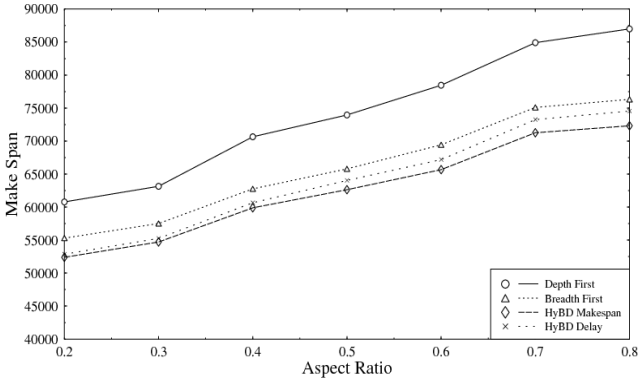


Figure 7(b). Make span vs. Aspect ratio

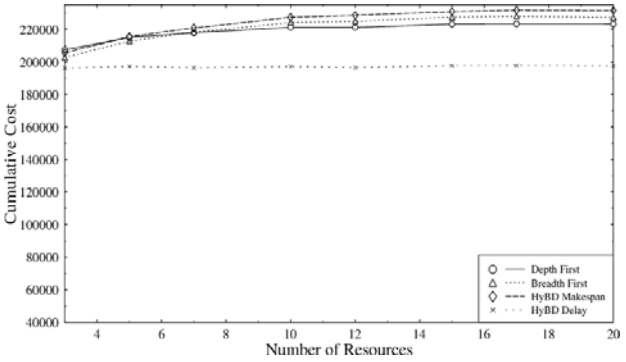


Figure 8(a). Cumulative cost vs. Number of resources

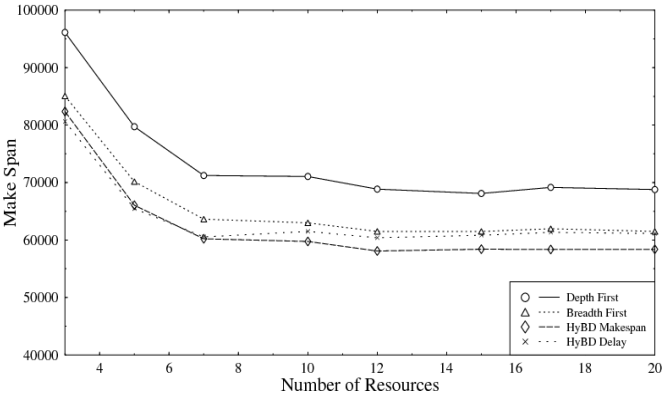


Figure 8(b). Make span vs. Number of resources

6.2.2 Streaming Factor

This parameter is a probability measure of the streaming between workflow components. A streaming value of 0 means no streaming. All temporally non-pipelined SDF models come under this category, where a child waits for the termination of its parent to start execution. On the other hand, a value of 1 corresponds to a typical PN model where all the nodes execute concurrently. A value of 0.5 represents a symmetrically modeled workflow.

Different combinations of *aspect ratio* and *streaming factor* generate various temporal and spatial concurrency patterns in the graph, which should be appropriately exploited by the heuristics for optimizing component allocation.

6.2.3 Number of Resources

The analysis of heuristics under limited resource conditions is necessary especially when the number of tasks is relatively higher than the number of accessible resources.

6.3 The Simulator

Analyzing the behavior of scheduling models on large, heterogeneous and dynamic platforms such as a Grid is extremely difficult. The reason is that such platforms are highly unpredictable, and performing a meaningful comparison between the results obtained from real test cases is difficult. Furthermore, to perform the scalability analysis of the heuristic, a setup involving a large number of network and computational resources is required. This makes simulation an appropriate and preferable choice for conducting experimental evaluation.

In order to perform such analysis, we have developed a discrete event simulator in which our proposed heuristics are integrated. In order to simulate the Grid behavior, we used the Network Weather Services (NWS) [33] traces taken from several different resources.

7. Experiments and Results

We conducted a large set of 3,920 experiments with graphs, obtained from different combinations of parameters, namely *streaming factor*, *aspect ratio* and the *number of resources*. Such a large number of test cases prevented us from being biased towards a specific heuristic. We compared the execution time and cumulative computational costs of the tasks against each of the 3 parameters.

7.1 Execution Time

This section compares the three parameters (plotted on x-axes) against the execution time (plotted on y-axis) of the experiment. Figure 6(b) shows a declination in the execution time as the streaming between the components increases. The decline is expected as a high level of streaming provided more chances for co-scheduling, resulting in a reduced value at the y-axis. In contrast, Figure 7(b) shows a rise in the overall execution time resulting from the reduced spatial concurrency. The spatial concurrency decreased as the aspect ratio was increased. On the other hand, Figure 8 (b) shows an asymptotic behavior in the curves as the number of resources increased. The execution time was initially high owing to the low resource availability, which limited the concurrent allocation of the components. However, as the number of resources approximated the number of components, a limiting behavior in the execution time can be observed. This means that the application performance became less dependent on resource availability and more on other parameters after a threshold value.

HyBD based heuristics, when compared with breadth first and depth first always performed better. However, HyBD_DELAY always performed worse than HyBD_MAKESPAN, owing to the delays that the former incurs to optimize CPU utilization, but CPU time is saved at this expense as described next.

7.2 Cumulative Cost

Ideally, if all the components were executed independently, then their cumulative cost would remain the same on individual resources. However, owing to the normalization effect explained earlier in Section 3.2, the cumulative cost of the application actually increased. This relationship is important in order to understand the analysis described in this section.

In Figures 6 (a), 7(a) and 8(a) the cumulative computational cost for HyBD_DELAY remained the same and always lower compared to other heuristics. This is because HyBD_DELAY maximizes the resource utilization by optimizing the allocation of downstream components based on their computational requirements. Thus, a downstream component gets delayed in cases where the upstream writer has more cost units than the reader. This essentially saves time that would have been wasted whilst waiting for data arrival. However, this is at the expense of additional delays in the total make span of the application as demonstrated in Figures 6(b), 7(b) and 8(b).

8. Conclusion and Future Work

In this paper we proposed GridRod, a new service oriented workflow-scheduling tool. GridRod integrates **Grid**dLeS and Nim**rod**/G machinery for providing the orchestration services based on web-and Grid-based architectures. A new heuristic model called HyBD is proposed which exploits both spatial and temporal concurrency between workflow components for scheduling. The scheduler, in this case, not only allocates workflow components to distributed grid resources but also specifies the communication mechanism for inter-component interaction. We suggest that the runtime communication specification gives the proposed model an edge over the currently existing workflow scheduling efforts. We leverage the already existing flexible IO mechanisms provided by GriddLeS to achieve such runtime flexibility. We believe that our work opens a new field of dynamic workflow modeling and scheduling on which further research is required. We also propose two novel heuristics called HyBD_MAKESPAN and HyBD_DELAY under the HyBD model as initial steps towards the new dynamic workflow-scheduling model. As our proposed model is unlike other prevalent approaches, we performed an empirical evaluation of our heuristics. The obtained results show that both heuristics behaved consistently with their respective objective functions whilst scheduling workflow components. HyBD_MAKESPAN reduced the application make-span in all the cases whereas HyBD_DELAY always reduced the cumulative cost unit of the application components, thereby saving a significant amount of CPU time.

Our future efforts include the improvement of the current scheduling model and investigation of advanced scheduling heuristics. The objective of this investigation is to perform efficient scheduling of applications with high communication to computation ratios (CCRs). Furthermore, we are interested in integrating GridRod with advanced workflow specification tools such as Kepler for more comprehensive workflow specification, driving us to develop more descriptive models of computation for workflow orchestration.

9. REFERENCES

[1] Abramson, D., et al., *High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid ?*, in *International Parallel and Distributed Processing Symposium*. 2000.

[2] Abramson, D. and J. Komineni, *A Flexible IO Scheme for Grid Workflows*, in *IPDPS-04*. 2004: New Mexico.

[3] Ilkay Altintas, A.B., Kim Baldridge,Wibke Sudholt, Mark Miller, Celine Amoreira,Yohann Potier and Bertram Ludaescher. *A Framework for the Design and Reuse of Grid Workflows*. in *Intl. Workshop on Scientific Applications on Grid Computing (SAG'04)*. 2005: Springer.

[4] *The Taverna Project*. [cited; Available from: http://taverna.sourceforge.net.

[5] *The Genie Project*. [cited; Available from: http://www.genie.ac.uk

[6] Anthony Mayer, S.M., Nathalie Furmento, Jeremy Cohen, Murtaza Gulamali, Laurie Young, Ali Afzal Contact Information, Steven Newhouse and John Darlington. *ICENI: An Integrated Grid Middleware to Support E-Science*. in *Workshop on Component Models and Systems for Grid Applications*. 2004. Saint Malo, France: Springer US.

[7] K. Seymour, H.N., S. Matsuoka, D. Dongarra, C. Lee, and H. Casanova, *GridRPC: A remote procedure call api for grid computing*, in *ICL Technical Report ICL-UT-02-06*. June 2002, Innovative Computing Laboratory, Department of Computer Science, University of Tennessee: Baltimore, MD, USA.

[8] *The VrGrads Project*. [cited; Available from: http://vgrads.rice.edu/.

[9] *The Kepler Project*. [cited; Available from: http://kepler-project.org/.

[10] Messerschmitt, E.A.L.a.D.G. *Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing*. in *IEEE Transactions on Computers*. Jan 1987.

[11] Thomas L. Adam, K.M.C., J. R. Dickson. *A comparison of list schedules for parallel processing systems*. in *Communications of the ACM*. 1974: ACM Press New York, NY, USA.

[12] Messerschmitt, E.A.L.a.D.G. *Synchronous Data Flow*. in *Proceedings of the IEEE*. 1987.

[13] Kahn., G. *The Semantics of a Simple language for Parallel Programming.* in *In Proceedings of IFIP Congress*. 1974: North Holland Publishing Company.

[14] Matias, P.B.G.a.Y. *New sampling-based summary statistics for improving approximate query answers*. in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 1998. Seattle, Washington, United States.

[15] Charu C. Aggarwal, J.H., Jianyong Wang, Philip S. Yu. *A Framework for Clustering Evolving Data Streams* in *In Proceeings of the 29th VLDB conference*. 2003.

[16] Liang Chen Reddy, K.A., G. . *GATES: a grid-based middleware for processing distributed data streams*. in *In Proceedings of IEEE Conference on High performance Distributed Computing, 2004. Proceedings*. 4-6 June 2004: IEEE Computer Society Press.

[17] Ahmad, Y.-K.K.a.I. *Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors*. in *ACM Computing Surveys*. 1999.

[18] Anthony Mayer, S.M., Nathalie Furmento, William Lee, Steven Newhouse, John Darlington. *ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time* in *Proceedings of the Workshop on Component Models and Systems for Grid Applications*. 2003. Saint Malo, France: SpringerLink.

[19] Ron Oldfield, D.K., *Applications of Parallel I/O* Oct 1996.

[20] Abramson, D., Kommineni, J., McGregor, J. and Katzfey, J. *An Atmospheric Sciences Workflow and its Implementation with Web Services*. in *The International Conference on Computational Sciences*. June 6 – 9, 2004. Krakow Poland.

[21] Jette., M.A. *Performance Characteristics of Gang Scheduling in Multiprogrammed Environments*. in *In Proceedings of the 1997 ACM/IEEE conference on Supercomputing*. Nov - 1997.

[22] Luiz Meyer, Mike Wilde, Marta Mattoso, Ian Foster. *Planning spatial workflows to optimize grid performance*. in *Distributed systems and grid computing (DSGC)*. 2006: ACM Press New York, NY, USA.

[23] Casanova, H., et al. *Heuristics for Scheduling Parameter Sweep Applications in Grid Environments*. in *In Proceedings of the 9th Heterogeneous Computing Workshop (HCW00)*. 2000.

[24] Casanova, H., et al., *The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid*, in *In Proceedings of the Super Computing Conference (SC'2000).* 2001.

[25] Abramson, J.K.a.D. *GriddLeS Enhancements and Building Virtual Applications for the GRID with Legacy Components*. in *European grid conference*. 2005. Amsterdam: Springer.

[26] Stiles, J.R., et al., *Monte Carlo simulation of neuromuscular transmitter release using MCell, a general simulator of cellular physiological processes.* Computational Neuroscience, 1998: p. 279-284.

[27] Foster, I. and C. Kesselman, *Globus: A Meta-computing Infrastructure Toolkit.* International Journal of Supercomputer Applications, 1997. **11**(2): p. 115-128.

[28] Hategan, M., et al., *GridAnt - A Client Controllable Grid Workflow System*. 2003, Argonne National Laboratory.

[29] Sarkar, V., *Partitioning and Scheduling Parallel Programs for Multiprocessors*. 1989: Paperback. 215.

[30] S. Cheng, J.S.a.K.R. *Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems*. in *Real-Time Symposium*. December 1986.

[31] H. Topcuoglu, S.H., and M.Y. Wu. *Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing*. in *IEEE Trans. Parallel and Distributed Systems*. 2002.

[32] Andrea C. Arpaci-Dusseau, D.E.C., Alan M. Mainwaring. *Scheduling with Implicit Information in Distributed Systems*. in *Joint Conference Measurement and Modeling Computer Systems*. 1998. Madison, Wisconsin.

[33] Python xml.dom

[34] *DAGMan (Directed Acyclic Graph Manager)*.

[35] Garg, A.a.R., Adrian *Straight-Line Drawings of Binary Trees with Linear Area and Arbitrary Aspect Ratio*. in *Proceedings Graph Drawing*. 2002. Irvine, CA, USA.

[36] Rich Wolski, N.T.S., Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing in Future Generation Computer Systems. 1998.

Nimrod/K: Towards Massively Parallel Dynamic Grid Workflows

§ David Abramson, § Colin Enticott and † Ilkay Altinas
{david.abramson, colin.enticott}@infotech.monash.edu.au, altintas@sdsc.edu

§ Faculty of Information Technology,
Monash University,
Clayton, 3800, Victoria, Australia

† San Diego Supercomputer Center,
9500 Gilman Drive, MC 0505
La Jolla, CA 92093-0505, USA.

Abstract

A challenge for Grid computing is the difficulty in developing software that is parallel, distributed and highly dynamic. Whilst there have been many general purpose mechanisms developed over the years, Grid programming still remains a low level, error prone task. Scientific workflow engines can double as programming environments, and allow a user to compose 'virtual' Grid applications from pre-existing components. Whilst existing workflow engines can specify arbitrary parallel programs, (where components use message passing) they are typically not effective with large and variable parallelism. Here we discuss dynamic dataflow, originally developed for parallel tagged dataflow architectures (TDAs), and show that these can be used for implementing Grid workflows. TDAs spawn parallel threads dynamically without additional programming. We have added TDAs to Kepler, and show that the system can orchestrate workflows that have large amounts of variable parallelism. We demonstrate the system using case studies in chemistry and in cardiac modelling.

1 Introduction

Grid computing has been proposed as the next generation of infrastructure to support distributed applications in science, engineering and business [13][14][20]. The Grid provides mechanisms that harness computational resources, databases, high-speed networks and scientific instruments, allowing users to build innovative virtual applications. Such virtual applications are synthesized by combining multiple different components on multiple computational resources.

A significant challenge for Grid computing is the difficulty in developing software that is concurrent, distributed and highly dynamic. There have been many mechanisms developed over the years for building such systems, ranging from remote procedure calls, general IPC mechanisms like TCP/IP sockets, parallel computing techniques based on message passing, through to recent work in Web services. However, programming still remains a low level and error prone task.

Recently, a number of different groups have developed scientific workflow engines that can double as programming environments for the Grid. These Grid Workflow systems [17][23][5][4][19][11][18][27] [10][12] [24][31][32][33][34] allow a user to compose a complex virtual application based on pre-existing, in some case, legacy components. In this model, components typically take input and produce output as part of a pipeline. The workflow system schedules the computations on the most appropriate (or selected) resource only when the inputs are available. Likewise, when the output is produced, it is forwarded to the next computation in the pipeline. Grid Workflows have been applied to diverse fields such as Computational Chemistry [5], Ecology [4] and Bioinformatics [19].

However powerful, current workflow engines are remarkably static – that is, the workflows themselves do not typically change at run time. Importantly, parallelism is typically specified statically when the underlying workflow graph is generated. Because of this, there are usually only two ways to specify parallel activity in a workflow:

1. Build a graph with multiple independent sub-graphs; or
2. Create a graph that contains logic to spawn parallel activities.

The first of these means that the amount of parallelism cannot change once the graph has started execution, which is restrictive. Further, if there is a lot of parallelism, the graph can become very large and difficult to manage. For example, a workflow that wishes to process the contents of a database, in parallel, must be replicated many times to allow them to run in parallel. The underlying graph can become very large, but it also contains a high degree of redundancy since the same operations are applied to each database element. This is the approach taken with tools like APST [8]. The second approach can support dynamic parallelism, but requires the user to write a very complex graph. For example, users must explicitly code for parallel execution, and incorporate loops that process the contents of the database, provide

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

mechanisms that spawn independent calculations and then synchronize the results. This process significantly complicates the task of writing a workflow. In spite of this, the approach is typical of systems like Kepler [17] [23] and Taverna [19], for example.

Over the years we have developed expertise in massively parallel parameter sweep workflows, using the Nimrod family of tools [3]. Nimrod is a system that makes it very easy to build parameter sweep and search applications. Users can run an arbitrary executable image, varying a number of parameters. Nimrod contains tools that perform a complete parameter sweep across all possible combinations (Nimrod/G), or search using non-linear optimization algorithms (Nimrod/O) [1] or experimental design techniques (Nimrod/E) [25]. Importantly, the number of jobs, and thus the parallelism, can be varied at run time, and the Nimrod scheduler places tasks on the available resources at run time.

However, Nimrod was not designed to execute arbitrary workflows of the type discussed above. Thus, it is difficult to run sweeps *over* workflows, and workflows *containing* sweeps. Likewise, as illustrated, most workflow systems do not support the parallel execution of tasks that is supported in Nimrod, and are not well suited to parameter sweeps and searches.

In this paper, we discuss an approach that allows workflows to spawn parallel threads *dynamically* without additional programming. Users write the simplest possible workflow that captures their business logic, and the underlying system provides mechanisms to replicate it, at run time, as required. This approach makes it possible to use the same workflow with one or a million data values, regardless of the nature of the logic. It allows a user to mix sweeps over workflows and workflows that contain sweeps.

We have implemented a prototype tool called Nimrod/K that demonstrates the new mechanisms. Nimrod/K is built on Kepler's runtime engine (Ptolemy) [18]. It uses a dataflow execution model that was originally developed for highly parallel dataflow computers in the 1980's, and this provides an extremely rich execution mechanism. It leverages a number of the techniques developed in the earlier Nimrod tools for distributing tasks to the Grid.

The paper begins with a discussion of existing workflow engines, and in particular, discusses Kepler (as a typical one). We then discuss the tagged dataflow architecture, independently proposed by groups at MIT [6] and Manchester [16], and show how this provides an ideal mechanism for executing dynamic workflows. We then discuss our prototype implementation, followed by a few case studies.

2 Grid Programming in Kepler

Kepler is typical of many modern Grid workflow systems. It allows scientists from multiple domains to design and execute scientific workflows. Scientific workflows can be used to combine data integration, analysis, and visualization steps into larger, automated "scientific process pipelines" and "Grid workflows" [23][5].

Kepler builds upon the Ptolemy II framework [18] developed at the University of California, Berkeley. Ptolemy II is a Java-based software framework with a graphical user interface called Vergil. The Ptolemy II project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components.

The focus of the Ptolemy II system is to build models of systems based on the assembly of pre-designed components. These components are called actors:
"An actor is an encapsulation of parameterized actions performed on input data to produce output data. An actor may be state-less or state-full, depending on whether it has internal state. Input and output data are communicated through well-defined ports. Ports and parameters are the interfaces of an actor. A port, unlike methods in Object-Oriented designs, does not have to have call-return semantics. The behaviors of a set of actors are not well-defined without a coordination model. A framework is an environment that actors reside in, and defines the interaction among actors." [18].

The interaction styles of actors are captured by Models of Computation (MoC). A MoC defines the communication semantics among ports and the flow of control and data among actors. "Directors" are responsible for implementing particular MoCs, and thus they define the "orchestration semantics" of the workflow. Simply by changing the director of a workflow, one can change the scheduling and overall execution semantics of a workflow, without changing any of the components or the network topology of the workflow graph. Two directors that are commonly used for Grid programming are the Process Networks (PN) and the Synchronous Data Flow (SDF) directors, which are based on Kahn Process Networks.

A Process Network is a directed graph, comprising a set of nodes (processes) connected by a set of directed arcs (representing FIFO queues). Each process executes as a standalone sequential program and is wrapped as a Ptolemy II actor. The one-way FIFO channels are used for the communication of processes and each channel can carry a possibly infinite sequence (a stream) of atomic data objects (tokens). Since channels have, in principle, unbounded capacity, writes

to channels are non-blocking, while reads are blocking [18]. The SDF domain is a dataflow-based execution model in which a sequential execution order of actors can be statically determined prior to execution. This results in execution with minimal overhead, as well as bounded memory usage and a guarantee that deadlock will never occur.

Kepler extends PtolemyII with a significant number of actors aimed particularly at scientific applications, e.g., for remote data and metadata access, data transformations, data analysis, interfacing with legacy applications, web service invocation and deployment, provenance tracking, etc. Kepler also inherits from Ptolemy the actor-oriented modeling paradigm.

In spite of its significant power, Kepler, and many other current workflow systems, do not support dynamic parallel execution. Thus, as discussed in the introduction, users must explicitly code a workflow to cause it to execute elements in parallel – either by replicating the workflow statically, or adding looping constructs that scatter and gather threads. Both of these techniques significantly complicate the workflow and obscure the underlying business logic. In the next section we discuss a tagged dataflow architecture, and show that it provides a much richer execution environment for parallel workflows.

3 The Tagged Dataflow Architecture

Arvind and Nikhil, at MIT, and separately Gurd and Watson at Manchester, proposed a multiprocessing architecture that supports parallel execution of instruction streams. Both of these architectures execute programs as "graphs" that represent a machine instruction sequence for a given program. The underlying model assumes that an instruction can 'fire' when it has its operands, unlike the conventional Von-Neumann model in which an instruction fires when selected by the program counter. In a dataflow machine, multiple instructions can fire concurrently, and provided they are scheduled to different processing elements, the program can be executed in parallel. A number of dataflow machines were produced, however, the additional hardware complexity made them uncompetitive with Von Neumann machines of the time.

Figure 1 illustrates the basic principles of a dataflow machine. Here, the node of graph represents an arbitrary instruction, and tokens contain the value of each operand. Instructions, or nodes, consume and generate tokens. In step (a) a token arrives on one input of an instruction. In step (b) a token arrives on the other input. In step (c) the instruction fires, consuming the tokens on the inputs, and generating token on the output. When connected together, these instructions can be used to implement arbitrary complex programs.

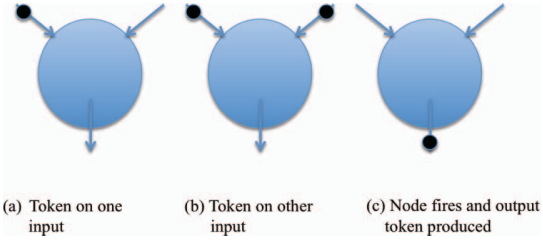


Figure 1 – instruction sequencing in a dataflow machine

In tagged token machines, tokens contain both a data field and a special tag field – or "colour", which is used to separate threads of execution. Importantly, an instruction fires when it has a token on each of its inputs that have the same colour values. Parallelism is implemented simply by creating tokens with different colours. Figure 2 shows an example. Here three tokens are stored on the left input of the instruction, each with a different colour. When tokens of the same colour arrive on the other input, the node fires. However, because each token has a different colour, three instances of the node execute in parallel. The output tokens are coloured the same as the inputs, and thus, as long as the colours remain distinct, the graph executes in parallel. It is worth noting that the machines include a number of colour manipulation instructions, and this makes it possible to both change the amount of parallelism dynamically, but also to change it in different parts of the graph. For example, it is not uncommon to spawn a number of concurrent threads for, say, parallel loops, and then collect the outputs back into a sequential instruction steam.

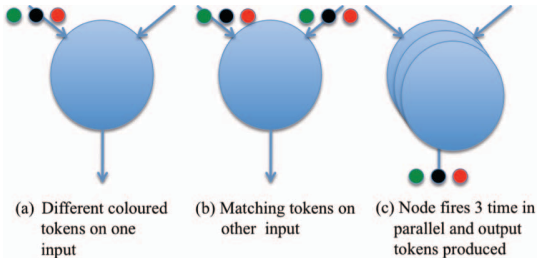


Figure 2 – instruction sequencing in a tagged dataflow machine

It is not possible to give a full description and all of the complexities of the instruction sets of the Manchester and MIT machines here. What is important is the idea of distinguishing concurrent threads of execution by colouring the tokens that flow through the graph. The architecture used in this work builds on a generalisation of the tagged architecture, and was proposed by Abramson and Egan [2]. This

generalisation allows tokens with the same colour to form a queue, and thus the model allows both queued (sequential) and tagged (parallel) execution.

As it turned out these techniques were not particularly successful for scheduling individual instructions at the machine code level, because the hardware and software overheads are substantially more than the cost of executing a single instruction. Thus, dataflow machines never achieved acceptance as a viable parallel architecture. However, the tagged dataflow idea can be used to support the type of dynamic parallelism discussed in this paper where the computations are much more significant than individual instructions, and thus the overhead is minimal. In the next section we will describe how we have used these ideas in the Nimrod/K tool set.

4 Design of Nimrod/K

In Section 2 we introduced the way Kepler (and the underlying Ptolemy) supports the use of different execution models with the same workflow. One of the reasons we have based our implementation on Kepler is that this important separation of concerns makes it possible to introduce a new execution mechanism based on the tagged dataflow execution model whilst leveraging the existing infrastructure. Accordingly, we have implemented a new Director called the Tagged Dataflow Architecture (TDA). The TDA builds on the existing SDF and PN directors but tags (or colours) tokens to distinguish different threads of execution. We envisage the idea could be added to other workflow engines with an ability to add new execution mechanisms. The TDA director supports existing 'legacy' actors in the PtolemyII project and the newer actors developed for Kepler, making a diverse range of actors available.

In the TDA, threads are identified by adding a unique tag to each token. These tag values are manipulated by a set of special tag manipulation actors, although the tag flow implementation is usually hidden from workflow designers. Even though some of Ptolemy II data types can be large complex structures, the implementation minimises memory usage by using pointers to the structure containing each token in the thread. This means the implementation scales well as the number of threads increases.

Underlying the SDF and PN directors are FIFO queues located on each of the inputs on an actor. Normally, when multiple tokens arrive on an input port, they are queued and can be processed when the actor is available. The TDA director follows the same procedure, but with a separate queue for tokens with different tags. This means that multiple tokens with the same tag value queue up, whereas tokens with different tag values can be consumed in parallel. Multiple actors cannot read from the same queue because they only read from queues with the same tag value assigned to them, and no two actors are given the same tag value. This gives the ability to copy actors (PtolemyII calls this cloning) and invoke them using different tags simultaneously. Importantly, this approach requires no changes to existing actors, which are usually unaware that they have been cloned.

We offer three methods to assign and modify tags within a workflow. The first two are aimed at the workflow *creator* who wishes to parallelize execution of a workflow. The third is aimed at the actor *developer* who wishes to develop new actors that support parallel threads.

The first method is transparent to workflow designers, and involves a number of new actors that generate and consume tagged tokens. For example, the Parameter Sweep actor shown in Figure 3 generates tokens that are already tagged. One could imagine a small set of such actors, such as reduction operators, etc. Clearly, this method is preferred because the parallelism is implicit in the workflow specification, and no explicit tag manipulation is required. Thus, this method makes it possible to take existing workflows and parallelise them without modification to the core logic. The second method provides a special tagging actor that adds a tag to a given token. This actor has two inputs, one of which takes a tag and the other takes an arbitrary token. The output of the actor is a token with its tag set.

The third method, which is directed at actor *developers*, provides an API that allows tagging, retagging or removing the tag of a token. This means that a developer can write complex actors that abstract tag management from the workflow creator, but still expose sophisticated thread management techniques. Likewise, we have provided two methods to access the values assigned to the tag from both the workflow design and from within an actor's code. First, when an actor is invoked, the tag value of its input tokens is added to the actor's parameter scope making it available in the actor's parameter options for a workflow creator to use. Second, we provide API functions so that actor developers can also access this information.



Figure 3 – a simple Kepler workflow

Figure 3, shows a simple parameter sweep experiment executing over an application called "My

program". In this example, the parameter sweep actor produces a number of tokens, each containing one of the parameter combinations (stored in a record). When using the TDA director, these tokens are also tagged with the value of the record. For example, if the value of the record is "X=1, Y=1", then the value of the tag is "X=1, Y=1". The "My program" actor accepts one token and produces one token as output, whilst the Display actor displays any tokens it receives. We will now compare the different behaviour of the SDF, PN and TDA directors as they orchestrate this simple workflow.

The SDF director [22] is a fully synchronised and pre-scheduling MoC. It executes its actors in turn until every actor reports it is unable to execute. The PN director, on the other hand, [22] activates every actor simultaneously and blocks an actor when there are no tokens available. Therefore, when executing the workflow in Figure 3 under the PN director, all actors are activated simultaneously and the "My program" and the Display actors are blocked immediately. As the parameter sweep actor produces tokens, the "My program" actor proceeds to process the tokens one at a time. Likewise the Display actor displays the tokens as they arrive. As it turns out, both the SDF and the PN directors behave in similar ways for this simple workflow.

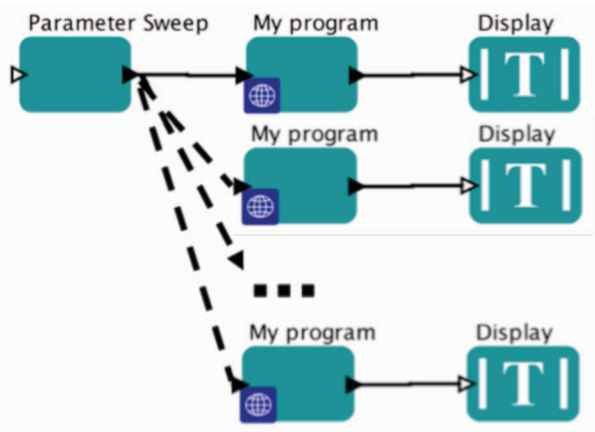


Figure 4 – The TDA director executing the simple workflow

Whilst the TDA director uses the same requesting technique as SDF, it also has the ability to create new threads for each of the tokens that it generates. In this example, the TDA director creates a single thread for the parameter sweep actor, which sequentially sends tokens to the "My program" actor. However, since each output token from the parameter sweep actor is tagged with a different value, multiple copies of "My program" execute in parallel. Importantly, the output of the "My Program" actor is tagged with the same value as its input, and thus, each "My program" thread communicates with a different Display actor thread. The effect of this is that the TDA director creates a separate thread for each of the parameter combinations and, as shown in Figure 4, they each display in parallel. It is important to highlight that Figure 4 depicts the *run-time* behaviour of the workflow, whereas, Figure 3 shows the workflow as *designed*.

A more complex workflow would probably gather or reduce the outputs of "My Program" together into a single thread, possibly reordering them into a single queue based on their tag values. This is analogous to a scatter-gather operation in a vector processor or a parallel loop instruction in a programming environment, and achieves the same sort of parallel execution.

Comparing directors shows the concurrency benefit of using the TDA director and how actors remain unmodified in all environments. Many of the Grid actors in Kepler do not provide information on the consumption and firing rate of tokens. In the example given in Figure 3, it may not be clear how many tokens the "My program" actor needs to consume nor how many tokens it will eject for each firing and if these numbers remain consistent. As discussed in [22], the SDF director determines a schedule for a workflow at the start of the execution. Because of this, it is unable to handle non-deterministic actors that change their token usage behaviour during the execution while the PN director adapts to the changing state. In this sense, the TDA director is similar to the PN as its schedule is not fixed.

State fullness is another attribute of an actor that needs to be considered. State fullness is where an actor holds information from a previous firing making the order of tokens entering the actor important. This can also happen indirectly through shared resources. These types of actors are safe using the SDF and PN directors as there is only one copy of the actor on the workflow. These actors are supported with the TDA director, but may be of limited value as they may not be able to be cloned because of concurrency issues, and thus, should be used with caution. The TDA environment allows both the actor developers and the workflow designers to specify if and when an actor can be cloned. Issues relating to state and parallelism will be discussed in further work.

5 Implementation Details

Figure 5 shows the structure of our prototype director implementation. There are three main components: the token scheduler, the actor manager and the director. The token scheduler's role is to decide which tag an actor should process next. The actor manager is

responsible for the execution of an actor, and for maintaining all of its copies. Like all directors in PtolemyII and Kepler, the TDA director maintains the execution of the workflow and reports to the workflow's calling function.

The token scheduler is responsible for deciding the actor firing order and the assignment of tokens to actor clones. The scheduler is notified of all token, movements and is responsible for deciding the number of actor copies that should be cloned at any time. This dynamic scheduling is different from the static scheduler used by the SDF director and the blocking method used in the PN director. We have also added a actor base class with an API that allows the scheduler to query the actor's resource requirements. A scheduler needs to consider resource availability to improve its scheduling even when using local resources. For example, a workflow may run a CPU intensive simulation package such as MATLAB which might be executed on a multi-processor. In this scenario, it is important for the scheduler to know how many MATLAB computations it can execute concurrently. Further to this, MATLAB might appear more than once in the workflow which needs consideration when scheduling these computations concurrently. It is the scheduler's responsibility to coordinate shared resources across the workflow.

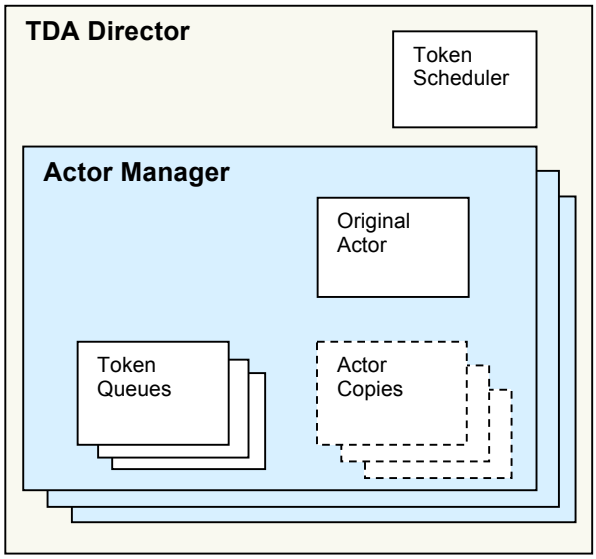


Figure 5 - Object associations

The prototype token scheduler is based on a simple outstanding request queue where an entry is added when a token arrives on any actors' input ports. The entry holds a reference to the actor and a reference to the token's tag's value, and ensures that all outstanding requests are processed before terminating the workflow. Storing references to these objects does not create inconsistency issues because actors are never changed during execution and tags are immutable in this environment. Only one outstanding request is kept per actor and tag combination to avoid unneeded multiple requests that will increase the queue size. This has the most noticeable effect when an actor requires more than one token to fire, but will only need a single firing request. Storing requests this way reduces the memory requirement of the queue because the actor only needs to be flagged once to execute. When an actor finishes firing and reports that it can fire again, the token scheduler decides whether to requeue this actor and tag combination by checking if there are any outstanding tokens. The only exception are actors that are defined as "source" actors which always generate a new request after a successful fire until the actor reports it is no longer able to fire. Our prototype token scheduler currently clones as many actors as required to consume all outstanding tokens at that time. However, this strategy may generate a large demand on resources including memory, in both the number actors and the number of active tokens, and on shared resources. Similar issues have been addressed in the past with the k-bounded loops work of Culler [9] and we plan to integrate some of these ideas in the near future.

Actor managers are responsible for maintaining the actors on the workflow. There is one actor manager for each actor on the workflow. They provide all the functions required to maintain an actor, including functions that manage actor cloning, token inputs and outputs, initialisation and cleaning up. Importantly, actors do not require modification to work under the TDA director because the actor manager performs all the functions required. These functions include the interface for a scheduler to query an actor's resource needs. The actor manager adds the tag's content (assigned by the token scheduler) to the actor's parameter scope and it ensures outgoing tokens have the correct tag for the actors that do not manage the tag itself.

The TDA director is invoked in the same way SDF and PN directors. Importantly, it is built on the same director interface as SDF and PN, and this allows it to interact seamlessly with the Ptolemy environment. It also has the functions to maintain synchronisation betweens the threads in its environment.

An important design feature of Kepler (and Ptolemy) is the use of multiple directors on the same workflow by using a hierarchical layout. This is implemented using special "Composite Actors" that supports nesting of directors. Just as PN directors allow an SDF controlled sub-graph actor in its workflow, a TDA controlled sub-graph can also be part of a PN controlled workflow. Further to this, it is possible to have an SDF sub-graph inside a TDA

workflow. There are rules governing which directors can be nested as a sub-graph on a workflow, which are explained in [15]. The TDA director closely resembles PN director on which combinations are possible with one exception, it is possible to have TDA controlled sub-graph as part of an SDF controlled workflow. This is because the TDA director does not block on reads and can release control back to the SDF director when no internal actor is active. To complicate the issue, placing a PN director inside a TDA controlled workflow blocks the TDA director from releasing control back to an outlying SDF controlled workflow. This is because PN directors do not relinquishing execution until it is told there are no more tokens at which point it terminates completely. We are currently exploring the semantics of nesting TDA with other directors and will produce a set of templates with clearly defined semantics for workflow developers.

6 Case Studies

In this section we demonstrate the effectiveness of our work using two case studies in theoretical chemistry and cardiac cell modelling. We have deliberately chosen fairly simple workflows as demonstrators in this paper to avoid complicating the description of the problem domains and the workflows. However, the same techniques have been applied to more complex examples in quantum chemistry, and will be reported in the future.

6.1 Case study 1 – Quantum Chemical Calculations

The first experiment concerns quantum chemical models – based on the Schrödinger equation – of assemblies of atoms as found in large biomolecules. To date, it has been impossible to use quantum methods on such large molecules because the computational time becomes prohibitive.

This work involves a set of approximations that replace large multi-atom systems with a single "pseudo" atom that has special theoretical properties that make it an analogue for the larger system. The solution involves computing a potential surface for the pseudo atom, called a pseudo-potential, and using this in the quantum calculations. A pseudo-potential does not describe a real chemical system, but can be used to approximate, and replace, a much larger molecular system. The result is that the quantum chemical calculations are much simpler, and faster, than if the original molecule had been used, opening the possibility of using quantum methods on very large systems. The details can be found in [28][29][30]. In our previous work we used Nimrod/G to schedule and execute the calculations.

The workflow shown in Figure 6 depicts the computation, which involves the execution of the GAMESS quantum chemistry package [26][21] a number of times, across a search space defined by 4 parameters (A, B, C and D). Here parameters A, B, C and D are to be determined that will provide the best fit between the real atomic system and target properties of the pseudo atom. The experiment used the results of the very large sweep over the A, B, C, D parameter space.

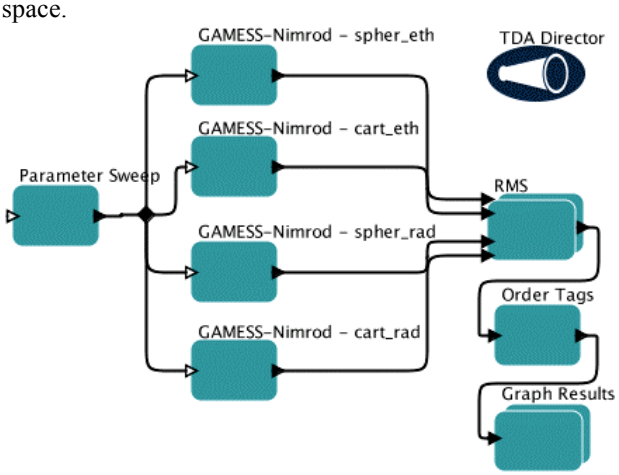


Figure 6 – GAMESS Workflow

The workflow shows 4 separate executions of GAMESS for each parameter set. The Parameter Sweep actor computes all combinations of the A, B, C and D, and builds a record for each of these combinations. They are used as inputs to the GAMESS actors, and the outputs are sent into an actor that computes the Root Mean Square error of the cost function. This is a measure of how well a particular pseudo potential surface fits calculations of a real molecule. The results are re-ordered and plotted using a Graph actor. Figure 7 shows the results across two of the parameters. The remaining parameter can be varied to produce multiple images, which can then be composed into a movie.

In the experiment we executed the workflow with three different Directors – SDF, PN and TDA. We highlight the simplicity of changing the execution semantics – all we had to do to change them was to swap out one director on the Vergil canvass and replace it with another one. Figure 8 shows the performance of the workflow under each of the Directors running on the testbed shown in Table 1. All compute resources have little communication latency (<1ms) and the data sets were very small and such had a negligible affect on the experiment time. The SDF director only executed one job at a time, as expected,

and took a total of 15:34:20. The PN director was able to exploit the static parallelism in the workflow, and executed up to 4 jobs at a time, taking 05:18:38. The TDA director was able to execute as many jobs together as there were processors available, and took a total of 00:29:19. Thus, the PN ran approximately 3 times faster than the SDF, and the TDA ran nearly 32 times faster than the SDF. The reason the PN times are not exactly 4 times faster is because the director waits on all 4 GAMESS-Nimrods to finish before submitting the next 4, and since each of them takes a different amount of time, time is lost in synchronising them. The graph in Figure 8 shows the number of jobs running at any time, and shows that the TDA Director peaks at 45 jobs. An initial observation is that the experiment used less than the maximum number of available processors (274) even though there were in principle sufficient jobs (484) to saturate the machines. This is because the jobs ran for only a few minutes allowing them to finish before all jobs were scheduled. Thus, the peak number of processors was only 45, and the average speedup only 32. We plan to optimize the job startup overheads in future implementations, and this will improve the speedup for experiments that have short running jobs.

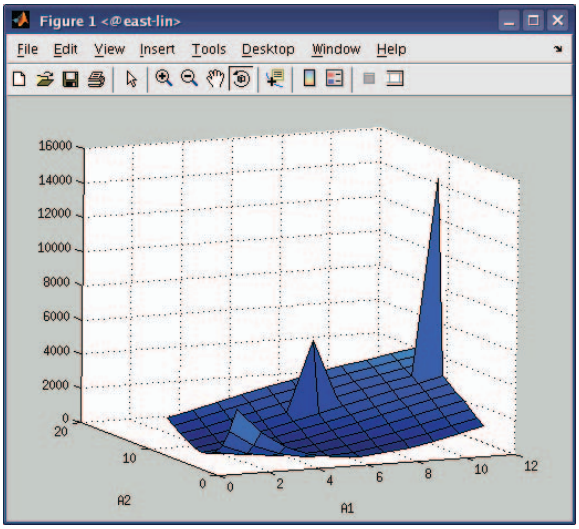


Figure 7 – Output of the GAMESS experiment

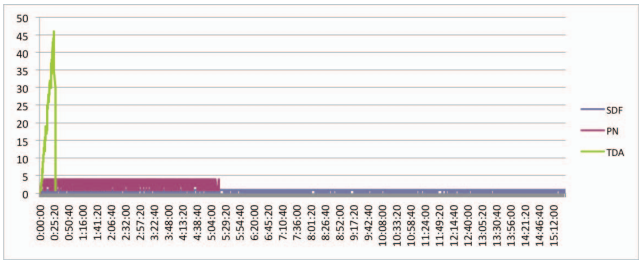


Figure 8 – performance results.

Machine	# Procs	Hardware	Operating System
East	160	Intel(R) Xeon(R) E5310 @ 1.60GHz	CentOS release 5
	64	Intel(R) Xeon(R) 5160 @ 3.00GHz	
Mahar	50	Intel(R) Pentium(R) 4 CPU 3.00GHz	Debian GNU/Linux 3.1

Table 1 – Testbed constitution

6.2 Case study 2 – Cardiac cell simulation

The second case study concerns a MATLAB model of a rabbit heart. This simulation solves a set of coupled ordinary differential equation that describe the various ion flows in a single cardiac cell. As in the previous case study, we wish to explore the behaviour of the model by varying a number of parameters, and then compare the ion levels (in particular the Ca^{2+} concentrations) with the results of real physiological experiment. [25]. An objective cost function, which indicates the solution quality for different parameters, is computed as moduli of the differences between computed metabolic-experimental values.

This case study highlights the TDA director's feature of accessing the tag's value and how it can be used to simplify workflows. We use the MATLAB actor packaged in the Kepler project to invoke MATLAB. Figure 9 shows the experiment workflow as set up for our new TDA director. The parameter sweep passes parameter values, as tokens with different tags, to the MATLAB actor. When executed under TDA, this causes multiple instances of the MATLAB actor to be cloned at run time. The TDA director adds the values of the parameters into the actor's "Parameter Scope" which is then substituted into the MATLAB script. However, with the SDF and PN directors, each parameter has to be passed into a parameter port, as shown in Figure 10, which is implemented with a CompositeActor. This CompositeActor only contains the MATLAB actor, which now has the parameters in scope, and an internal SDF director. This new way of adding parameters to actors using tag tokens makes the construction of workflows simpler. These methods of passing the parameters to the actor differ from the GAMESS case study which used Nimrod to pass the parameters into the script.

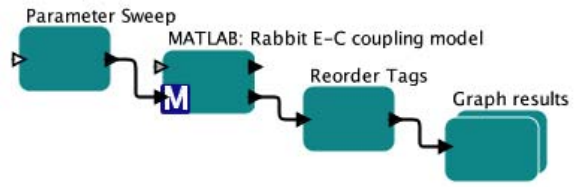


Figure 9 – MATLAB workflow with the TDA director

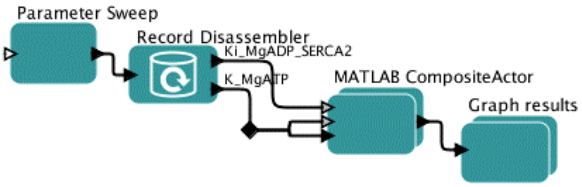


Figure 10 – MATLAB workflow with an SDF or PN director

This experiment was performed on a server that has two Intel(R) Xeon(R) E5310 @ 1.60GHz Quadcore processors allowing 8 processes to run simultaneuously. Using the SDF and PN directors, as expected, all the MATLAB jobs ran sequentially for 1:43:26 and 1:44:27 respectively. With the TDA director, all 64 MATLABs ran concurrently finishing the experiment in 16:54, delivering a speedup of 6.1. The value of the objective cost function is graphed against two of the parameters in Figure 11. Interestingly, this graph was produced using a standard Kepler actor for MATLAB, demonstrating the significant functionality available in an existing workflow system.

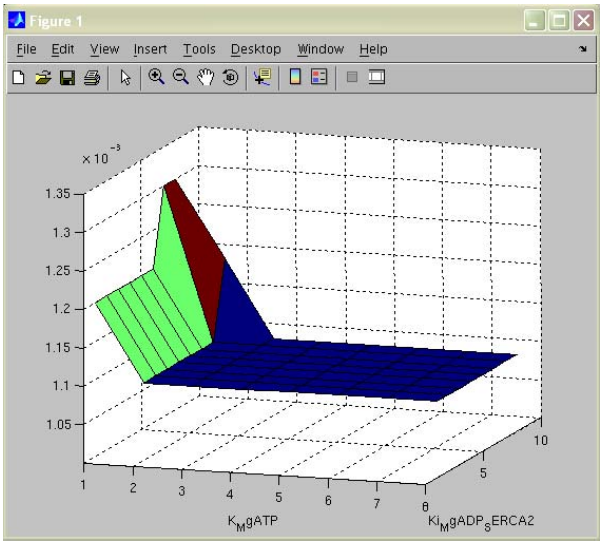


Figure 11 – Output of the MATLAB experiment

7 Conclusions

In this paper we have discussed the design and implementation of a new orchestration mechanism for scientific workflows. The scheme is based on mechanisms that were originally designed for parallel dataflow computers. When used in this mode, they allow a workflow to dynamically spawn threads of computation transparently from the workflow design, and this greatly simplifies the work for the user over existing workflow systems.

We have built a prototype version in Kepler, and have been able to reuse all of the existing Kepler code by simply adding a new director. This is very attractive because it now only allows us to reuse an existing workflow system, but also all of the components (actors) that have been developed to date.

We have illustrated the effectiveness of the scheme using two examples, both of which had been implemented using our Nimrod software previously. The new system is significantly more flexible and allows workflows of arbitrary complexity to be implemented.

The current system is only a prototype, and has illustrated the functionality and potential for our approach. However, systems like Kepler, allow components to pass more complex data types like files. File transport between Grid nodes adds a level of complexity and we will explore this in the near future. Further, the scheduling system in the current prototype does not have the sophistication of some of the other techniques we have implemented in Nimrod. We plan to explore more complex scheduling approaches, such as those discussed in [7].

Acknowledgements

This project is supported by the Australian Research Council. Thanks go to Wibke Sudholt from the University of Zurich for the GAMESS experiment discussed in first case study, and Saleh Amirriazi and Anuska Michailova, from the University of California at San Diego, for the second case study.

References

[1] Abramson D, Lewis A, Peachey T, Fletcher, C., "An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics", SuperComputing 2001, Denver, Nov 2001

[2] Abramson, D and Egan, G, "The RMIT Dataflow Computer: A Hybrid Architecture", The Computer Journal, Vol 33, No 3, June 1990, pp 230 - 240.

[3] Abramson, D., Giddy J., and Kotler, L. "High Performance Parametric Modeling with Nimrod/G:

Killer Application for the Global Grid," In *Int'l. Parallel and Distributed Processing Symposium (IPDPS),* Cancun, Mexico, May 2000. http://www.csse.monash.edu.au/~davida/nimrod/.

[4] Altintas, I. Berkley, C. Jaeger, E. Jones, M. Ludäscher B. and Mock, S. "Kepler: Towards a Grid-Enabled System for Scientific Workflows," in the *Workflow in Grid Systems Workshop in GGF10* - The 10th Global Grid Forum, Berlin, March 2004.

[5] Altintas, I., Birnbaum, A., Baldridge, K., Sudholt, W., Miller, M., Amoreira, C., Potier Y. and Ludaescher, B. "A Framework for the Design and Reuse of Grid Workflows" *Intl. Workshop on Scientific Applications on Grid Computing* (SAG'04), LNCS 3458, Springer, 2005.

[6] Arvind, and Nikhil R.S. "Executing a Program on the MIT Tagged-Token Dataflow Architecture", IEEE Transactions on Computers, Vol. 39, No. 3 (1990)

[7] Ayyub, S. and Abramson, D. "GridRod - A Service Oriented Dynamic Runtime Scheduler for Grid Workflows". 21st ACM International Conference on Supercomputing, June 16-20, 2007, Seattle.

[8] Casanova H. and Berman, F. "Parameter Sweeps on The Grid With APST", chapter 26. Wiley Publisher, Inc., 2002. F. Berman, G. Fox, and T. Hey, editors.

[9] Culler. D. "Managing Parallelism and Resources in Scientific Dataflow Programs", PhD thesis, Department of Electrical Engineering and Computer Science,Massachusetts Institute of Technology, March 1990.

[10] Deelman, E. Blythe, J. Gil, Y. Kesselman, C. Mehta, G. Vahi. K. "Mapping Abstract Complex Workflows onto Grid Environments", Journal of Grid Computing, 1:25-39, Kluwer Academic Publishers, Netherlands, 2003.

[11] e-Science Grid Environments Workshop, e-Science Institute, Edinburgh, Scotland, May 2004, http://www.nesc.ac.uk/esi/events/.

[12] Fahringer, T. Jugravu, A. Pllana, S. Prodan, R. Jr, C. S. and Truong H. L., "ASKALON: a tool set for cluster and Grid computing", Concurrency and Computation: Practice and Experience, 17:143-169, Wiley InterScience, 2005.

[13] Foster I. and Kesselman, C. "Globus: A Metacomputing Infrastructure Toolkit," *Int'l J. of Supercomputer Applications*, vol. 11, no. 2, 1997, pp. 115-128.

[14] Foster I. and Kesselman, C. (editors), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, USA, 1999.

[15] Goderis, A., Brooks, C., Altintas, I., Lee, E., Goble. C. "Composing Different Models of Computation in Kepler and Ptolemy II". 2007 Proceedings, International Conference on Computational Science (ICCS), April, 2007.

[16] Gurd J.R. and Watson I. "Data Driven System for High Speed Parallel Computing (1 & 2) Computer Design, vol.9 nos.6 & 7, June & July 1980, pp. 91-100 & 97-106.

[17] http://kepler-project.org

[18] http://ptolemy.eecs.berkeley.edu/ptolemyII/ptIIfaq.htm

[19] http://taverna.sourceforge.net

[20] http://www-unix.globus.org/toolkit/

[21] http://www.msg.ameslab.gov/GAMESS/

[22] Lee, E. et al, "Overview of the Ptolemy Project," Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March 6, 2001.

[23] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., Tao J. and Zhao, Y. "Scientific Workflow Management and the Kepler System", *Concurrency and Computation: Practice & Experience*, Special Issue on Scientific Workflows, 2005.

[24] Oinn, T. Addis, M. Ferris, J. Marvin, D. Senger, M. Greenwood, M. Carver T. and Glover, K. Pocock, M.R. Wipat, A. and Li. P. "Taverna: a tool for the composition and enactment of bioinformatics workflows", Bioinformatics, 20(17):3045-3054, Oxford University Press, London, UK, 2004.

[25] Peachey, T. C., Diamond, N. T., Abramson, D. A. Sudholt, W., Michailova, A. Amirriazi, S., "Fractional Factorial Design for Parameter Sweep Experiments using Nimrod/E", to appear, Journal of Scientific Programming.

[26] Schmidt, M. W., Baldridge, K. K., Boatz, J. A., Elbert, S. T., Gordon, M. S., Jensen, J. H., Koseki, S., Matsunaga, N., Nguyen, K., Su, S., Windus, T.L, Dupuis, M. and Montgomery, J. A. *General atomic and molecular electronic structure system,* J. Comput. Chem., v. 14, 1993, pp. 1347-1363;

[27] Scientific Data Management Framework Workshop, Argonne National Labs, August 2003. http://sdm.lbl.gov/~arie/sdm/SDM.Framework.wshp.htm

[28] Sudholt, W., Baldridge, K. K., Abramson, D., Enticott C. and Garic, S. *Applying Grid Computing to the Parameter Sweep of a Group Difference Pseudopotential,* in Computational Science – ICCS 2004: 4th International Conference, Kraków, Poland, June 6-9, 2004, Lecture Notes in Computer Science, v. 3036, 2004, pp. 148-155.

[29] Sudholt, W., Baldridge, K. K., Abramson, D., Enticott C. and Garic, S. *Parameter Scan of an Effective Group Difference Pseudopotential Using Grid Computing*, New Generation Computing, v. 22, 2004, pp. 125-136.

[30] Sudholt, W., Baldridge, K., Abramson, D., Enticott C. and Garic, S. *Application of grid computing to parameter sweeps and optimizations in molecular modeling*, Future Generation Computer Systems, v. 21, 2005, pp. 27-35.

[31] Tannenbaum, T. Wright, D. Miller, K. and Livny. M. "Condor - A Distributed Job Scheduler." Beowulf Cluster Computing with Linux, The MIT Press, MA, USA, 2002.

[32] Taylor, I. Shields, M. and Wang. I. "Resource Management of Triana P2P Services", Grid Resource Management, Kluwer, Netherlands, June 2003.

[33] von Laszewski, G. Amin, K. Hategan, M. Zaluzec, N. J. Hampton, S. and Rossi. A. "GridAnt: A Client-Controllable Grid Workflow System", In 37th Annual Hawaii International Conference on System Sciences (HICSS'04), Big Island, Hawaii: IEEE CS Press, Los Alamitos, CA, USA, January 5-8, 2004.

[34] Yu J., and Buyya, R., A Taxonomy of Workflow Management Systems for Grid Computing", Journal of Grid Computing, Springer Press, New York, USA.

Parameter Space Exploration using Scientific Workflows

David Abramson[1], Blair Bethwaite[1], Colin Enticott[1],
Slavisa Garic[1], Tom Peachey[1]

[1] Faculty of Information Technology,
Monash University,
Clayton, 3800, Victoria, Australia
{david.abramson, blair.bethwaite, colin.enticott,
slavisa.garic, tom.peachey}@infotech.monash.edu.au

Abstract. In recent years there has been interest in performing parameter space exploration across "scientific workflows", however, many existing workflow tools are not well suited to this. In this paper we augment existing systems with a small set of special "actors" that implement the parameter estimation logic. Specifically, we discuss a set of new Kepler actors that support both complete and partial sweeps based on experimental design techniques. When combined with a novel parallel execution mechanism, we are able to execute parallel sweeps and searches across workflows that run on distributed "Grid" infrastructure. We illustrate our new system with a case study in cardiac cell modelling.

Keywords: distributed workflows, parameter exploration, Kepler

1 Introduction

In recent years there has been a great deal of interest in "scientific workflows" [10]. These allow scientists to specify large computational experiments involving a range of different activities, such as data integration, modelling and analysis, and visualization. Activities can be composed, often using a graphical programming environment, so that the output of one stage can be passed as input to the next, forming a pipeline of arbitrary complexity. Scientific workflow engines manage the execution across a range of distributed resources, and leverage Grid computing middleware and approaches [1][2][3]. For example, it is possible to extract data from a scientific instrument, pass it through some analysis software running on a high performance cluster, store the results in a distributed data repository, and then visualize it on a large display wall. Scientific workflows have been used to great effect in a number of different disciplines including Computational Chemistry [7], Ecology[8] and Bio informatics [9].

Importantly, many workflow engines double as programming environments for the Grid. Whilst there are no standard ways of doing this, a number of engines effectively expose the Grid middleware APIs. For example, Kepler [5] exposes a

variety of middleware layers, from Globus through to ad-hoc interfaces like SSH. Other engines, such as Triana [19] and Taverna [9] allow users to invoke services as Web Services, but provide no explicit support for Grid middleware.

Prior to the wide adoption of workflow engines, we developed a family of tools, called Nimrod, for performing parameter sweeps with computational models. Nimrod supports the execution of a specific type of workflows in which a single computation is performed multiple times to allow exploration of some design space. Nimrod includes tools that perform a complete parameter sweep across all possible combinations (Nimrod/G) [4], or search using non-linear optimization algorithms (Nimrod/O) [20] or experimental design techniques (Nimrod/E) [**Error! Reference source not found.**]. Importantly, the degree of parallelism, can be varied at run time, as the Nimrod scheduler places tasks on the available resources then. Nimrod had been applied to a wide range of disciplines from public health policy to quantum chemistry [22].

However, Nimrod was not designed to execute arbitrary workflows of the type discussed above. Thus, it is difficult to run sweeps *over* workflows, and workflows *containing* sweeps. Likewise, most workflow systems do not support the parallel execution of tasks that are supported in Nimrod, and are not well suited to parameter sweeps and searches.

In this paper we discuss how we have added parameter sweeps and searches to existing workflow tools. We focus on the techniques that allow a range of scenarios to be explored by adding a few simple components to an existing workflow. Specifically, we discuss the design of a new family of Kepler "actors" that support sweeps across parameter ranges. The resulting system is extremely flexible, and allows the creation of decision support systems of arbitrary complexity.

The paper begins with a discussion of parameter sweeps and the techniques that are commonly used, illustrating the ideas through the Nimrod family of tools. We then discuss workflow engines, and in particular, we describe Kepler. We then introduce a new family of actors, and show how these can be combined with existing workflows. Finally, we demonstrate the applicability of the ideas using a case study in cardiac modelling.

2 Parameter Sweeps

2.1 Full Parameter Sweeps

Many current scientific and engineering problems can be formulated as computational models with a great deal of accuracy. Changing the inputs allows a user to explore a range of design scenarios, giving a picture of how the system behaves. This is typically performed as a sweep over the input parameters. Although the model may be computationally expensive, parallel execution of the jobs can dramatically speed up the execution and allow very large systems to be studied.

In this context, it is useful to think of a given computational model as a function that accepts a set of input parameters and produces a set of outputs. Input parameters are typically simple types, such as integers, floats and text strings. A full parameter

sweep takes all combinations of the parameters, and allows exploration of the entire design space within some finite resolution. Such parameter sweeps have been used very effectively, for example in environmental modelling [23], bioscience [24], engineering [**Error! Reference source not found.**] and chemistry [25].

2.2 Partial Parameter Sweeps

Although the favoured method for exploring the parameter space of a model is a full parameter sweep, this may be impractical where there are many parameters, especially if the model is computationally intensive. Experimental design techniques may enable meaningful results to be obtained from a suitably chosen subset of jobs.

The most widely used method of experimental design is the fractional factorial design [11]. Suppose the experiment has input parameters (called factors in the literature) *A, B, C,* … and produces a numerical output *f(A,B,C…)*. The underlying model is that *f* is the sum of several terms, a constant term f_0, then the "main effect" terms $f_1(A)$, $f_2(B)$, …, each dependent on only one factor, then "two-factor interaction effects" terms, $f_{1,2}(A,B)$, $f_{1,3}(A,C)$, $f_{2,3}(B,C)$, …, each dependent on two factors, then "three-factor interactions", and so on. Long experience, with both physical and computer-based experiments, suggests that the higher-order terms, interactions of three factors and more, are usually negligible. In that case, only the lower-order terms need to be estimated, so fewer jobs are needed to obtain these estimates and hence obtain reasonable approximations to the output. Significant savings are possible; with 20 parameters, each at two levels, the number of jobs required is reduced from over a million to 512.

2.3 The Current Nimrod Tool Set

Over the past 15 years we have constructed a tool set called Nimrod [4], that automates both of the parameter sweep techniques discussed above. One of the Nimrod tools, Nimrod/G, supports complete parameter sweeps It operates either as a tool, or a middleware layer in its own right. If the former, then it is usually operated from a Web portal, and this allows a user to create a plan file, set up a testbed, manage certificates etc., and organize input and output files from an experiment. Nimrod/G, however, can also serve as a job management system for other software, including the other members of the Nimrod family. This structure is shown in Fig. 1.

The Nimrod/E tool, on the other hand, automates the design of fractional factorial experiments. Here, the user specifies the factors and which interactions can be ignored. Then, one component produces an efficient design generating the parameter values for the resulting jobs in a form suitable for Nimrod/G. When all jobs are complete, another component produces analyses of the results for each output value of interest. It creates graphs showing the relative size of the various main effects and interactions. The Daniel Plot [13], Fig. 6, plots the effects so that the negligible ones, with values just experimental error, will form the central straight line, significant effects produce points that deviate from this line. The Lenth Plot [14], Fig. 5, shows

effects in order of absolute size and horizontal lines giving significance levels. Points outside the outer lines are most probably significant. The tool also estimates results of a full parameter sweep, which may then be used by visualization software.

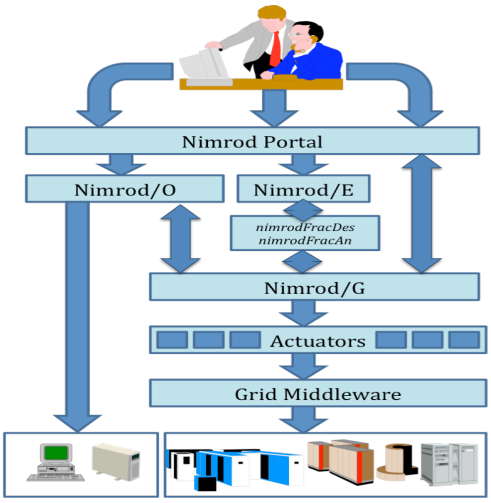


Fig. 1. Nimrod tool chain

3 Workflow Engines & Kepler

There has been considerable interest in scientific workflow engines over the past few years, in particular, in Grid workflow engines. Workflows allow a user to build arbitrary computations from a set of connected components, or *actors*. Actors may represent computations, but also facilitate access to distributed databases and scientific instruments. In most systems, data moves along the edges that connect the actors as *tokens*, and can thus be streamed from instruments and databases, through a range of computational processes. Grid workflows allow these actors to be executed on distributed resources, and launched in a variety of ways, facilitating virtual applications that span multiple organizations, data sources and computers.

To date, many scientific workflow tools have been built [6][9][10]. Grid workflow systems allow general applications to be constructed, with examples ranging from ecology to medical imaging. In this project we focus on Kepler [5][6][7][8], which allows a user to weave a "virtual" application from a set of otherwise distinct components, and its workflow engine orchestrates their execution in a controlled and repeatable manner. Kepler builds upon Ptolemy II [11], a Java-based software framework with a graphical user interface called Vergil. Ptolemy II is used for the modelling, simulation, and design of concurrent, real-time embedded systems. The focus is on assembly of concurrent components. Kepler inherits a number of orchestration mechanisms from Ptolemy, providing an extensive range of execution mechanisms. These are controlled through devices called "directors".

In spite of their significant power, Kepler, and many other current workflow systems, do not support dynamic parallel execution of a workflow and its components. This means that users must explicitly code a workflow to cause it to execute elements in parallel – either by replicating the workflow statically, or adding looping constructs that scatter and gather threads. Both of these techniques significantly complicate the workflow and obscure the underlying business logic. In another recent paper, we have shown how to augment Kepler with a Tagged Data Flow Architecture Director (TDA) [18]. This new system, called Nimrod/K, extends Kepler by providing powerful mechanisms for exposing and managing parallelism in the workflows. This provides an ideal platform for using workflows for parameter sweeps. Unlike the current Nimrod tool chain, Nimrod/K makes it possible to run sweeps over workflows, and workflows that contain sweeps. It leverages Kepler's power in building complex workflows, and Nimrod's ability to execute sweeps over grid resources. In the next section we show how we have created some specific new Kepler actors that facilitate parameter sweeps. When combined with Nimrod/K's parallel execution mechanisms, we have a powerful new tool for parameter sweeps.

4 New Kepler Actors

4.1 Actor Design

The Nimrod/G Actor. As discussed in section 2.3, the current Nimrod tools provide parameter sweep tools to complement Nimrod/G's Grid execution and meta-scheduling capabilities. One of these tools performs a parameter sweep that creates a Grid job for each of the parameter combinations in the parameter space. Each of these parameter combinations is represented in Kepler using datasets, called "tokens". We have created a `ParameterSweep` actor, that uses the same parameter syntax as the Nimrod/G implementation, to generate parameter combinations in a workflow. This actor works with the current Ptolemy directors, however, when used in combination with the Tagged Dataflow mechanism in Nimrod/K, parallel execution of the different parameter combinations allows the workflow to execute efficiently on the Grid.

The new `ParameterSweep` actor is quite general. For example, multiple actors can be chained together to create a sub parameter sweep inside a larger parameter sweep. Although it is possible to create such a sweep with a single `ParameterSweep` actor, staging them allows for preliminary calculations of shared data to be performed by the parent sweep prior to the start of the sub-sweep. Also, chaining them together enables a more dynamic parameter range options, for example, if the sub sweep is calculated from the parameters given by the parent sweep. This environment produces sweeps that meet the parameter sweep requirements of workflows.

Nimrod/E Actors. The workflow for a typical Nimrod/E experiment is shown in Fig. 2. The user enters details of the parameters to be varied and the effects to be estimated, into the `FractionalFactorialDesigner` actor. This designs the

experiment and specifies the jobs required, passing these job specifications (as workflow tokens) to the actor(s) that perform the model execution. If the model is a simple formula then the model execution may be performed by a Kepler numerical actor. More complex modelling may require execution of an external model via the Nimrod/K framework.

When all jobs are complete, downstream processing can begin. There may be several numerical aspects of the model output of interest, so at this stage the dataflow may branch. Fig. 2 shows the case where there are two outputs requiring analysis. One output from each model is needed to estimate the effects for that output. Tabulation of the effects may be sufficient, or these estimates may be used as inputs for a Daniel plot, Lenth plot and/or computation of the full sweep results.

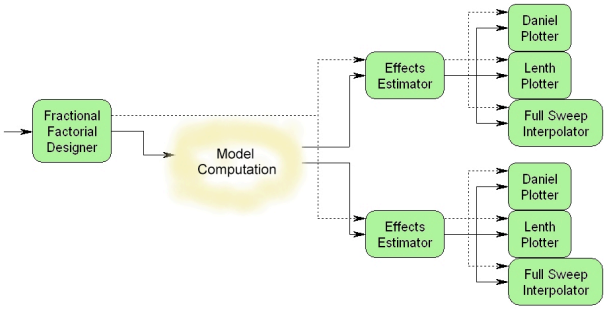


Fig. 2. Nimrod/E experiment

4.2 Implementation

Nimrod/G. The *ParameterSweep* actor is built on the PtolemyII source class that provides an output port and a trigger. The output port sends a RecordToken containing the value of each parameter for a single parameter combination. The actor has a property that contains the parameter sweep string identical to the syntax of the parameter sweep commands in the Nimrod plan file syntax. To implement the token tagging functions, the *ParameterSweep* actor is built on the classes and functions provided by the Nimrod/K TDA director. These classes have no functional affect when used with other Ptolemy directors. Under the Nimrod/K director each parameter combination is executed in parallel. This is further discussed in [18].

Nimrod/E. The actors for Nimrod/E were built using the same principles as the *ParameterSweep* actor. Because *nimrodFracDes* is computationally intensive there is a clear advantage in retaining the C code for this function. Further, the code is quite complex, so that re-writing in Java within the Kepler framework would be a large undertaking. Consequently, we decided to use the existing C code, using a Java wrapper to produce a design actor. The information that it produces as a plan file is redirected as a tag to the output token for downstream processing. This actor is named the *FractionalFactorialDesigner* in Fig. 2.

For the analysis section most computation is relatively light. Further, the graphs produced would benefit from a re-coding in Java, using the Kepler graphing functionality. When there are a large number of factors, the plots produced may display estimates for many effects and be quite cluttered; Kepler graphs provide zooming to allow the user to explore the fine detail. On the other hand, the production of the analysis matrix does require some of the complex computation used in `nimrodFracDes`. Accordingly, we decided to recode the analysis section within the Kepler framework. The `EffectEstimator`, `LenthPlotter`, `Daniel-Plotter` and `FullSweepInterpolator` actors were added to perform the analysis discussed in Section 2.3.

5 Case Study

In this section we illustrate the new actors we have discussed to date, using a case study previously performed with the Nimrod/G and Nimrod/E tools. The case study allows us to demonstrate the utility of the new approach and its expressive power in Kepler.

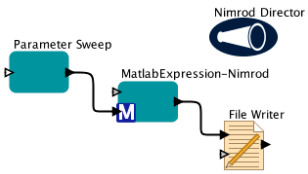


Fig. 3. Complete parameter sweep

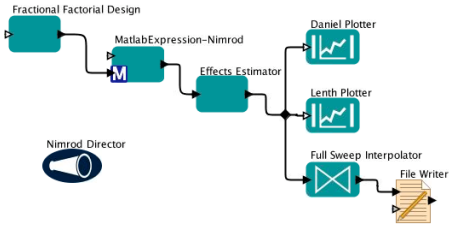


Fig. 4. Experimental Design

Mathematical models of the heart show considerable promise for understanding the underlying mechanisms and for clinical diagnosis and treatment [15][12]. The model chosen [16] concerns excitation and contraction in rabbit ventricular muscle cells. Intracellular flows of calcium, sodium and magnesium ions were modelled as a system of ordinary differential equations using Matlab. Earlier experiments with this model are reported in [17]. Fig. 3 shows the Kepler workflow used to perform a parameter sweep over the nine factors, *A* to *J*., This used two values for each factor, producing an experiment of 512 jobs, and the values are generated using the `ParameterSweep` Actor.

A second experiment used the experimental design functionality to further investigate the response function. The `FractionalFactorialDesign` actor in the workflow shown in Fig. 4 was used to produce a design that would estimate the main effects and two-way interactions of these factors. This required only 128 jobs.

Fig. 5 shows the Lenth plot from the experiment. Points for the effects of all nine inputs lie outside the confidence lines and hence all inputs make a significant contribution. Of the interactions, only, *AB, DE, GH, GJ* and *HJ* are definitely significant, the other interactions are indistinguishable from noise. This reveals the

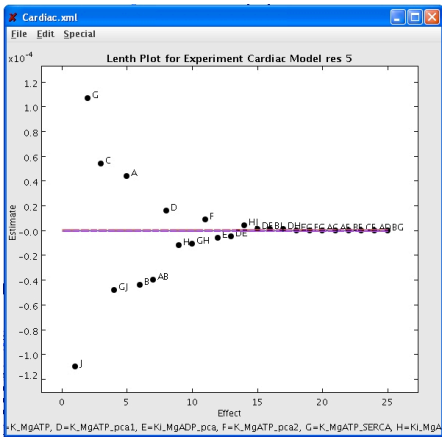


Fig. 5. Lenth plot

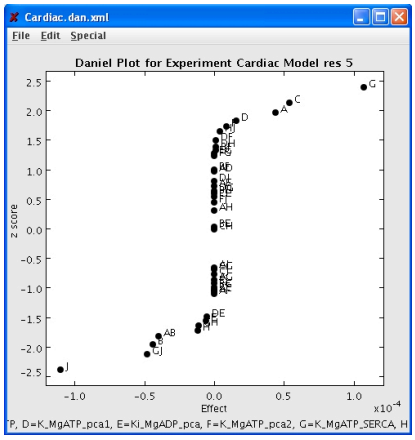


Fig. 6. Daniel plot

structure of the computational model. The final output is the sum of four terms, the first affected only by *A* and *B*, the second by *C*, the third by *D*, *E* and *F* and the last by *G*, *H* and *J*, explaining why many interactions have no effect. However the lack of significant interactions between *D* and *F*, and between *E* and *F*, requires further investigation. Fig. 6 shows the same results on a Daniel plot. Here the deviation of points from the vertical line gives a measure of the significance of the effects plotted.

Fig. 7 shows the parallelism in the complete sweep when executed under the Nimrod/K Director. Our cluster was saturated at just over 100 concurrent executions. The system takes some time to build up the parallelism, and this is evident in the initial ramp up phase. This is caused by the overheads in starting remote computations using the Globus framework.

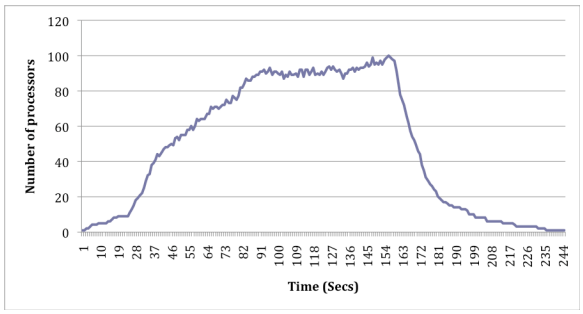


Fig. 7. Parallelism in complete sweep

This case study demonstrates that Kepler provides a natural and easy mechanism for specifying the use of parameter sweep techniques over existing workflows. We have used a fairly simple computation model in this paper to clearly show the

techniques, but this does not really demonstrate the significant potential of the approach. All of the machinery developed to date is capable of sweeping across parameter combinations regardless of the complexity of the computational steps of the workflow. Thus, much more complex pre-existing workflows can be modified using our new actors to perform large-scale complex parameter sweep experiments. We have recently begun work on modifying a complex computational chemistry workflow by adding the actors discussed in this paper.

6 Conclusions

In this paper we have discussed the design and implementation of some new actors that facilitate parameter sweeps in scientific workflows. The solution builds on the existing execution frameworks in workflow systems, and we have demonstrated its applicability in Kepler. The simple case study shows how easy it is to create and execute a sweep over an existing scientific computation.

Our system leverages a separate addition to Kepler, called Nimrod/K, which supports parallel execution of the different instances. By combining the new Actors, and the new TDA Director, we can execute each of the parameter combinations in parallel. This makes it possible to compute complex design experiments quickly if there are sufficient resources available.

7 Acknowledgements

The Nimrod/K project is supported by Australian Research Council ARC Discovery grant. We are grateful to our colleague, Ilkay Altintas, from the San Diego Supercomputing Center for her assistance, and Anushka Michailova and Saleh Amirriazi, from the UC San Diego, for their work on the cardiac model.

References

1. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
2. Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. Int'l J. of Supercomputer Applications, vol. 11, no. 2, pp. 115--128. (1997)
3. The Globus Toolkit: http://www-unix.globus.org/ toolkit/
4. Abramson, D., Giddy, J., Kotler, L.: High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid. In: Int'l. Parallel and Distributed Processing Symposium (IPDPS), Cancun, Mexico (2000)
5. Kepler Project, http://kepler-project.org
6. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., Tao, J., Zhao, Y.: Scientific Workflow Management and the Kepler System. In: Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows (2005)

7. Altintas, I., Birnbaum, A., Baldridge, K., Sudholt, W., Miller, M., Amoreira, C., Potier, Y., Ludaescher, B. A Framework for the Design and Reuse of Grid Workflows. In: Intl. Workshop on Scientific Applications on Grid Computing (SAG'04), LNCS 3458, Springer (2005).
8. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludäscher, B., Mock, S., Kepler: Towards a Grid-Enabled System for Scientific Workflows. In: the Workflow in Grid Systems Workshop in GGF10 - The 10th Global Grid Forum, Berlin (2004)
9. Taverna Project, http://taverna.sourceforge.net
10. Yu, J., Buyya, R. A Taxonomy of Workflow Management Systems for Grid Computing. Technical Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, Univ of Melbourne (2005)
11. Box, G. E., Hunter, W. G., Hunter, J. S. Statistics for Experimenters. Wiley (1978)
12. Amirriazi, S., Chang, S., Peachey, T., Abramson, D., Michailova, A. Optimizing Cardiac Excitation-Metabolic Model By Using Parallel Grid Computing. Biophysics 2008, Long Beach, California (2008)
13. Daniel, C. Applications of Statistics to Industrial Experimentation. Wiley (1976)
14. Lenth, R. V. Quick and Easy Analysis of Unreplicated Factorials. Technometrics 31, pp 469-473 (1989)
15. Hunter, P. J., Kohl, P., Noble, D. Integrative Models of the Heart: Achievements and Limitations. Philosophical Transactions of the Royal Society of London: Mathematical, Physical and Engineering Sciences, 2001, 10.1098/rsta.2001.0816 (2001)
16. Michailova, A., Lorentz, W., McCulloch, A. Modelling Transmural Heterogeneity of KATP current in Rabbit Ventricular Myocytes. AJP-Cell Physiology, 293, C542-C557, (2007)
17. Amirriazi, A., Chang, S., Peachey, T., Abramson, D., Michailova, A. Optimizing Cardiac Excitation-Metabolic Model By Using Parallel Grid Computing. Biophysics 2008, Long Beach, California, February (2008)
18. Abramson, D., Enticott, C., Altinas, I. Nimrod/K: Towards Massively Parallel Dynamic Grid Workflows. IEEE Supercomputing 2008, Austin, Texas, November (2008)
19. Taylor, I. Shields, M. and Wang. I. "Resource Management of Triana P2P Services", Grid Resource Management, Kluwer, Netherlands, June 2003.
20. Abramson, D, Lewis A, Peachey T, Fletcher, C., An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics, SuperComputing 2001, Denver, Nov (2001)
21. Peachey, T. C., Diamond, N. T., Abramson, D. A. Sudholt, W., Michailova, A. Amirriazi, S., Fractional Factorial Design for Parameter Sweep Experiments using Nimrod/E. Journal of Scientific Programming, Volume 16, Numbers 2,3 (2008)
22. Message Lab list of current and former e-Science projects, http://www.messagelab.monash.edu.au/EScienceApplications
23. Lynch, A. H., Abramson, D., Beringer, K. J., Uotila, P. Influence of savanna fire on Australian monsoon season precipitation and circulation as simulated using a distributed computing environment. Geophys. Res. Lett., 34, L20801, doi:10.1029/2007GL030879 (2007)
24. Sher, A., Abramson, D, Enticott, C, Garic, S, Gavaghan, D., Noble, D., Noble, P., Peachey, T. Incorporating local Ca2+ dynamics into single cell ventricular models using Nimrod/O. ICCS 2008, Krakow, Poland, June (2008)
25. Sudholt, W., Baldridge, K., Abramson, D., Enticott, C., Garic, S. Parameter Scan of an Effective Group Difference Pseudopotential Using Grid Computing. New Generation Computing, 22 pp 125-135 (2004)

Virtual Microscopy and Analysis using Scientific Workflows

§ David Abramson, § Blair Bethwaite, § Minh Ngoc Dinh, § Colin Enticott, ‡ Stephen Firth, § Slavisa Garic,
‡ Ian Harper, ‡ Martin Lackmann, § Hoang Nguyen, § Tirath Ramdas, § A.B.M. Russel, Ω Stefan Schek, ‡ Mary Vail

§ Faculty of Information Technology, Monash University, Clayton, 3800, Victoria, Australia

‡ Faculty of Medicine, Monash University, Clayton, 3800, Victoria, Australia

Ω Leica Microsystems Am Friedensplatz 3, Mannheim, D68165, Germany

Abstract - Most commercial microscopes are stand-alone instruments, controlled by dedicated computer systems. These provide limited storage and processing capabilities. Virtual microscopes, on the other hand, link the image capturing hardware and data analysis software into a wide area network of high performance computers, large storage devices and software systems. In this paper we discuss extensions to Grid workflow engines that allow them to execute scientific experiments on virtual microscopes. We demonstrate the utility of such a system in a biomedical case study concerning the imaging of cancer and antibody based therapeutics.

I. INTRODUCTION

Since the advent of the microscope, the development and sophistication of imaging technologies have set the pace in life-sciences, and the ability to "distinguish" smaller and smaller structures continues to define the foundation for biological, bio-medical and bio-technological research. Emerging ultra-high resolution microscopes now allow us to monitor individual molecules in the context of a living cell [23]. Already, off-the-shelf technologies provide real-time, non-invasive images of microscopic bio-medical phenomena, such as the binding of anti-cancer drugs to individual tumour cells, or the trafficking of individually-marked metastasizing tumour cells in and out of blood vessels [13]. However, we expect that both the spatial and temporal resolution of the instruments, and the associated data volumes and rates, will increase dramatically over the next 5 to 10 years. Moreover, our understanding of the underlying cell biology will demand increasingly sophisticated image processing and data mining techniques.

Currently, commercial microscopes are predominantly stand-alone instruments, controlled by dedicated computer systems that can only provide limited storage and processing capabilities. Routinely, the experimenter interacts directly with the microscope, which captures and stores digital images on a local computer disk. The data is then transferred manually to various computer and storage devices for analysis, mining, archiving and visualization. Increasing data volumes are already beginning to create a bottleneck in which the processing times exceed the data acquisition times, significantly impeding real time image visualization and interpretation. Modern image analysis techniques (such as volume rendering), and data mining software, require multi-processor systems for adequate performance and large data stores for file storage, but these require a range of ad-hoc and complicated methods involving meta-data capture, file transport, job submission and data archiving and visualization.

To address this challenge a number of groups have discussed the seamless integration of image capturing hardware and data analysis software into a wide area network of high performance computers, large storage devices and software systems, generating a 'virtual' multi-modal instrument [15][17][5][21]. Such federation supports a variety of different experimental modes, allowing data to be captured on remote instruments, processed using appropriate high performance computers, displayed on a variety of imaging devices and archived on remote storage servers.

Together with our colleagues at Leica, we are building such a Grid environment, depicted in Figure 1. We have implemented a control system that uses a scientific workflow engine to script, and then execute, the experiments using a graphical programming environment. This overall approach has many advantages; most notably the use of a single representation to both document, and execute, an experiment – for the first time, the traditional laboratory notebook becomes executable. Also, because many workflow tools now incorporate the ability to archive, and even share, workflows [16][3], the approach provides a very robust working environment.

However, simply having a workflow engine is not enough to realise the vision. We also need the ability to control and capture images, process them on parallel supercomputers, access network attached storage and display results on visualisation devices.

Workflow tools, such as Kepler, have components that support some of these functions [7]. For example, Grid execution 'actors' support remote computation; SRB 'actors' provide access to remote Storage Resource Broker services; and local computation actors support simple image manipulation. However, we require additional components to support remote control of microscope hardware, acquisition of images, display of images on a variety of devices, as well as techniques for easily exploiting parallel image processing pipelines. Further, some steps require human guidance and interpretation, so it is impossible to fully automate the image pipeline. To facilitate this we require techniques for involving users in key processing steps within the workflow framework.

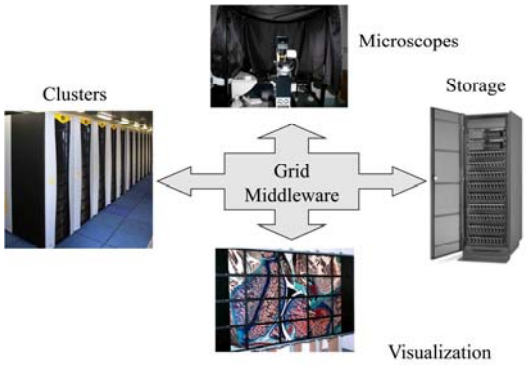


Figure 1 – a Grid enabled virtual microscope

In this paper, we discuss the design and implementation of new Kepler actors that achieve these goals. Specifically, we have built:

- actors for accessing microscopes regardless of their physical location;
- actors that display workflow results on tiled display walls (e.g. OptiPortal [4]);
- actors that incorporate user actions into the workflow.

We also leverage our prior work in providing dynamic parallelism in Kepler, to enable high throughput transparent processing of images using Grid connected parallel machines.

To demonstrate the feasibility of such an approach, we describe an experiment driven by bio-medical science, in which images are captured by microscopes in Australia and Germany, and then processed, displayed and archived in Australia. We leverage the benefits of large tiled display walls by aggregating a number of images, and involve the user in helping evaluate which ones are of interest. The case study highlights the significant advantages in building on open software tools, and illustrates that our approach is flexible and powerful.

II. GRID WORKFLOWS AND KEPLER

Grid workflows allow a user to build arbitrary computations from a set of connected components, or *actors*. Workflows enable applications that control instruments, access and archive data, process the data on high performance computers, display the results on novel visualisation devices and allow the users to interact with their computations. In short, workflows are the scripting language that facilitates the rapid prototyping and execution of novel applications of the type being considered here. Actors may represent computations, but also facilitate access to distributed databases and scientific instruments.

Grid workflow systems allow general applications to be constructed, with examples ranging from ecology to medical imaging. Whilst there are many experimental and commercial workflow systems (see [22] for a fairly recent review paper), in this paper we focus on Kepler [14][7], an open multiparty collaboration. Kepler allows a user to weave a 'virtual' application from a set of otherwise distinct components, and its workflow engine orchestrates their execution in a controlled and repeatable manner. Many Kepler features make it an ideal engine for orchestrating microscopy experiments. For example, Kepler:

- provides a rich set of built-in actors for accessing data and databases;
- provides mechanisms for executing processing logic on local and remote computers;
- provides methods for building web service client-side code, allowing arbitrary web services to be integrated into computations;
- supports a range of different execution semantics, allowing business logic to be separated from performance considerations; and
- is extensible, allowing users to add functionality as required.

On the other hand, it is quite difficult to express workflows with dynamic parallelism using Kepler. This complicates parallel execution of image processing operations, as discussed in the paper. Previously, we have developed a system that augments Kepler with a new *Director* that makes it much easier to specify such experiments [2]. This new system, called Nimrod/K, extends Kepler by providing powerful mechanisms for exposing and managing parallelism in the workflows. In this paper we show how we have used this to process images in parallel, accelerating image throughput.

We have also built a new set of actors that allow workflows to access remote instruments, incorporate user input, and display results on high resolution display devices, such as the OptiPortal. These will be discussed in the next section.

III. WORKFLOW EXTENSIONS

A. New Actors

1) Web service instrument control

Many modern instruments expose their control systems using techniques ranging from direct IP sockets with proprietary protocols through to standard approaches such as RPC, RMI, COM and DCOM. These mechanisms provide a way of controlling the instrument remotely, opening up significant scientific potential as discussed in section 1.

We have used Web services over these mechanisms because they provide an open and extensible interface that can be accessed by a variety of different tools and frameworks. For example, the Kepler workflow engine discussed earlier provides mechanisms for accessing Web services, and thus we can control the microscope from a workflow.

In addition to the backend sever side functionality, we have implemented four specific Kepler actors for controlling Leica microscopes. These actors can be used standalone, as part of a larger workflow, or within composite actors. Specifically, they are:

`StartScanning`: This actor starts the microscope image scanning process using default experimental settings. It checks hardware parameter settings such as focus, objective position, laser, intensity, channel and others before starting image scanning. This process also triggers the conversion of images into different formats, and stores

images at a defined location in the local file system of the microscope (or in a shared file system if one is present). The raw images are generated sequentially channel by channel.

`SetDataExporter`: This actor sets the default image export URL. This URL is used by the `StartScanning` actor to store images at a defined location. This could be on a network-attached storage based medium, or on a local microscope disk, in which case the file needs to be exposed using some protocol such as FTP, as this then allows the workflow to access the data.

`Execute`: This actor loads the pre-defined experiment parameters with optimal hardware settings for any experiment. Usually, it is set by an operator, who places a sample, focuses the microscope and stores experiment parameters into an Instrument Parameter Setting (IPS) file. The IPS file consists of all microscope hardware settings that are specific to certain image capture methods.

`StopScanning`: This actor stops the experiment on the microscope. This actor is important for long running experiments where smart scripting can be used to stop the running experiment and change position of a sample. This mode supports "what if" exploration, and is essential for live cell imaging where cells can move out of focus.

Security is an important consideration in attaching expensive and sensitive instruments to the Grid. There are a number of security models that could be used, however, many of these were unavailable because of security constraints at Leica. For example, WS-Security – the obvious model to be used in conjunction with the Web service interfaces discussed still requires a Web server to be exposed to the Internet. However this requirement clashed with Leica's security policy so the two were incompatible. Accordingly, we accessed the microscopes via a VPN connection, in which Leica's corporate network was mounted on the controlling workstation. This allowed us to authenticate using Leica's approved security mechanism, and the Web server was then exposed using a local IP address. The approach is a little restrictive because it is only possible to access one site in this way. If we needed to VPN to more than one site, the technique would not have worked. An alternative is to use tunnelling techniques, of the type discussed in [20].

2) Displaying results

While microscope images can be rendered using conventional display technology, the output becomes unwieldy when there are either multiple, or single very high resolution, images. For example, image processing operations may generate multiple results (as will be illustrated in our case study), and we may wish to compare and contrast them visually. Alternatively, modern microscopes can scan a large sample, producing tiles that are subsequently stitched into a single large image. Such a requirement demands more screen "real estate" than available on a traditional display.

Recently, DeFanti and others developed the OptiPortal [4], a cluster based system that links many independent monitors to produce a single very large display. The OptiPortal is an ideal display for showing the output of either a single high resolution image, or multiple independent images. However, to date, there is no easy method for connecting scientific workflows to the OptiPortal, and thus, this is currently a manual and error prone task. Typically, a user must copy the images to the OptiPortal head node, and then use a special windowing environment to display them. This requires the control of multiple systems, and becomes complex when images are produced as a stream.

Accordingly, we have developed a new actor that takes an image object and renders it on the OptiPortal. The actor, `OptiPortalDisplay`, consists of a Java client, modelled on the Kepler `ssh` actor, that connects to the front end of the OptiPortal. It then invokes a special server program, and establishes a direct TCP/IP socket connection. Images are streamed from the client as they are computed, and displayed on the Optiportal. We have based our implementation on the CGLX windowing environment [9], and will expand this to the SAGE platform in the near future [10]. The server currently supports the display of static images and videos, however, we envisage more complex data objects being supported. For example, it should be possible to send a VRML object for display and manipulation using the same actor.

3) Users as actors

Human decision-making capacity is difficult to replace in many applications, including medical imaging. For this reason, it is often necessary to incorporate human feedback within workflows. There are two basic requirements for such functionality: (1) a user interface to allow humans to enter information, and (2) a mechanism to convey this information back into the workflow environment. Furthermore, it is desirable that the user interface is capable of presenting rich information, and also that it is possible to present the interface to a user in a remote location.

Web technology is one implementation approach that can meet all the above basic and desired requirements. Using this approach, a workflow can generate a webpage that presents information in a meaningful manner to aid the decision-making process, and this webpage would convey the user decision through communication with a `WebListener` actor. In the present prototype implementation, the `WebListener` actor binds to a pre-determined port and waits for a HTTP GET request, then conveys user input into the workflow either through the return value (if user input is limited to a small integer range) or through standard output (if user input is less constrained). It is necessary to launch the web interface and `WebListener` in parallel since both processes must run in order for the complete user interface mechanism to work, and this is achieved by running them under Kepler's Process Networks *Director* (PN).

This mechanism is powerful, and sufficient to support the functionality required for the workflows presented in this paper. Furthermore, the simplicity of this approach allows a large degree of flexibility in the kinds of interactions that may be supported – the tradeoff being that the workflow designer has the burden to ensure that: (1) the webpage interface suits the target application with

the constraint that the user feedback may be expressed as a single HTTP GET request; and that (2) once this information has been conveyed into the workflow, the workflow will process this HTTP GET command and process the information appropriately. Of course, the Web browser used for the interaction can be viewed on the same OptiPortal that shows the images, further enhancing the interaction.

B. *Parallel image processing in Kepler*

At a functional level, it is possible to specify an arbitrary image processing pipeline using Kepler (and other workflow engines). Operators can either be invoked as local functions (coded in Java) or by remote execution actors using Web services, `ssh` or Grid primitives such as `globus-job-run` (for execution on a single Grid resource) or `nimrod-run` (for meta-scheduling on a set of Grid resources) [1][11].

Running this pipeline in parallel becomes more difficult because Kepler does not provide high-level support for parallel execution of the workflow. Recently we proposed a set of extensions to Kepler, called Nimrod/K, that allow a functional pipeline to run in parallel with minimal changes [2]. Nimrod/K adds a new Kepler *Director* that supports multi-threading. Data tokens are augmented with a tag, or colour, field. When an actor consumes these tokens, a new thread is initiated for each new colour. For example, 100 tokens are sent into an actor, each with a different colour field, then 100 different instances of the actor execute concurrently. If the actor performs a local image processing operation, then 100 processes are forked concurrently on the machine on which Kepler is executing. This would only be efficient if that machine was a symmetric multiprocessor. However, it is also possible to use a remote invocation actor as discussed, in which case, the function can be launched on remote resources, such as Grid connected clusters.

In this paper we will illustrate the combination of the Nimrod/K Director with the `nimrod-run` actor to launch multiple image processing operations on remote clusters, leveraging Nimrod's ability to balance the load across multiple machines.

IV. CASE STUDY

A. *Cancer research and imaging*

Recently-emerging concepts in tumour therapy are based on the understanding that tumour growth, invasion and metastasis relies on blood vessels for the supply of nutrients, and on connective tissue to provide scaffolding for the tumour cells and blood vessels [12][18][19]. While monoclonal antibodies that target key signalling molecules are already approved anti-cancer agents, their mode of action in tumour vessels is controversial and the subject of intense research.

In this case study we use fluorescent-labelled antibodies together with reagents marking tumour stroma and tumour vasculature to examine the targets of this potential anti-cancer therapeutic. In particular, we use a variety of microscopes and imaging techniques to address a range of research needs.

A "birds-eye' view of the complete, 6 μm-thick tumour section, compiled using the tile scan function of a Leica AF6000LX microscope to image Hoechst-stained cell nuclei and lectin-stained blood vessels, reveals a well-defined tumour centre with tightly-packed tumour cells which is surrounded by extensive vascular network feeding into the tumour. In order to assess if the antibody therapy reaches the vascular and stromal targets within the tumour, we need high-resolution images in selected areas of the tumour section, using a confocal point-scanning microscope. In this case, a Leica TCS SP5 housed at Leica Mannheim in Germany. 2D analysis at four different wavelengths of a selected tumour area within the same section reveals at subcellular resolution the accumulation of the antibody to endothelial cells lining blood vessels and to the connective tissue.

To complete the analysis, we capture on the SP5 a complete optical 'Z-stack' from a 60 μm thick section of the same tumour and use this to recreate a high resolution 3D model of the antibody-targeted tumour vasculature. Before the 3D image can be rendered, it is necessary to 'clean' the data. However, choosing the correct threshold values typically requires user input. The way this is normally performed is for the scientist to apply a threshold, then render the image, and observe if it appears to be correct. If not, then they modify the threshold value and iterate until the image is acceptable. This process can be quite time consuming because a full 3D rendering (taking about 32 minutes) is required at each iteration. As an alternative, we chose to apply a range of different threshold values, render the cleaned images in parallel, and present a range of 3D visualizations (actually as movies) to the operator, who then chooses the most appropriate. The application of high throughput computing techniques reduces the critical path for the operator, at the expense of some additional computation. This approach only works because we have sufficient processors to perform the rendering, and sufficient screen real estate to display the images in one step – making comparison easy.

Overall, the simultaneous display of lower-resolution views of the whole tumour section with high-resolution analyses of areas of interest on a multi-panel screen provides the experimenter with images of unprecedented clarity and resolution. This approach enabled findings about the distribution of the tumour targeting antibody IIIA4 mAb within a tumour – in this particular experiment – with a new level of confidence.

B. *The Testbed*

As discussed, we make use of two different microscopes, both enabled with the Web Service interface discussed in 3.1.1. Specifically, a Leica AF6000LX wide field microscope at Monash University, Melbourne, Australia and a Leica TCS SP5 confocal microscope at Leica, Mannheim, Germany are used to capture images.

The parallel image processing is performed on a distributed Grid of Linux clusters at various sites around Melbourne. Images are displayed on a 4 x 4 (32 MPixel) OptiPortal at Monash University.

C. *The experiment*

Here we show a number of Kepler workflows, and describe how these implement the experiment discussed in 4.1. We highlight the use of the new actors and parallel processing capabilities discussed in section 3. In many cases, in order to simplify the discussion in the paper, we have specified high-level *compound* actors that abstract irrelevant detail.

Workflow 1 (Figure 2) shows the logic required to capture the tile-scan from the AF6000LX microscope in Melbourne. The `StartScan` composite actor generates 30 images (two channels by 15 different positions), which are merged to form a two-color 3x5 tile-scan, covering an area of 2 mm x 4.5 mm of the sample. The workflow copies two directories of files from the microscope (each one containing all positions for that channel), merges them into 15 two colour tiles, stitches the tiles together (using `Tile Placement`) in a pre-defined manner and produces a final bitmap image. It then displays this result on the OptiPortal (as shown in Figure 3).

Within this bird's-eye view, one can identify regions of interest for further study (shown by the white circle in Figure 3). A magnified section from within the circle is shown in Figure 4, where the green pixels show a cross section of a blood vessel.

The image in Figure 4 clearly warrants a higher resolution scan rather than a simple magnification of the original tile scan. This functionality is incorporated into Workflow 2 (Figure 5), which displays the image in a web browser with Javascript code to identify the coordinates of a user click within the image. It then transmits these coordinates as a HTTP GET request through a HTTP submission form. This HTTP GET is received by the `WebListener` (discussed in 3.1.2), providing the x and y coordinates selected by the biologist. The coordinates are used to trigger a high-resolution multi-channel scan at the required location. This time, the scan is performed on the SP5 microscope in Germany (using a shipped slide), and the data is retrieved by FTP as before. Importantly, the only change required to the workflow to reflect the different location of the microscope concerns information in the IPS file, discussed in 3.1.1 and the file names for the FTP. The workflow downloads four separate files (one for each channel), and merges these into two separate colour images – one with the Blue and Green channels, and one with all colours. The resulting images are displayed on the OptiPortal for examination.

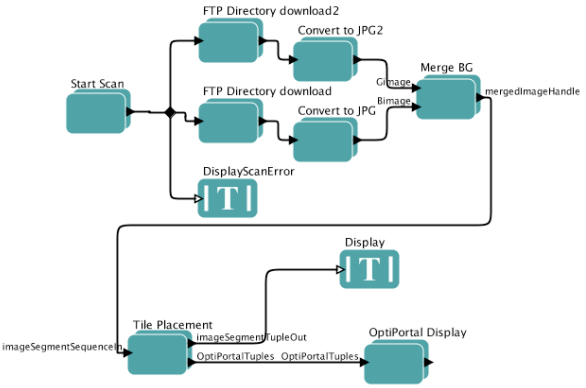


Figure 2 – Workflow 1 – Bird's-eye capture and display

Based on the analysis of the higher resolution scan, shown in Figure 6 (blood vessels, green (RCA-lectin-FITC); tumour targeting antibody, red (labelled with quantum dots)), we then perform a three dimensional z-stack acquisition at the region of interest, again using the SP5 confocal microscope. Workflow 3 (Figure 7) contains the logic discussed in 4.1. Here, 64 different thresholds are applied to the stack to eliminate noise. This is performed as a parameter sweep across the threshold value using ImageJ, an open image processing toolkit commonly used in biomedical research [8]. Since this operation is so fast, we run these on the local workstation. Each of the filtered images is then rendered from 360 single degree perspective views using VolumeJ [6]. This latter operation is performed as a parameter sweep generating 360 different images in total. The resulting 360 * 64 (23,040) images are combined into a movie that shows the results of 64 different threshold values, rotated through 360 degrees, as the movie proceeds. These are presented to a human operator to select the best candidate, and correspondingly the best threshold value.

In order to produce the movie quickly, we leverage the clusters shown in Table 1. The Nimrod/K Director and the `NimrodRun` actor are used to parallelise this fairly simple pipeline. Because of the short execution time for individual rendering jobs, we batched them into 20 degree fields, generating a total of 1152 jobs. These were distributed across 517 processors over three Grid resources (East, South and West as shown in Table 1). Using the resources we were able to reduce the rendering time from nearly 34 hours to about 6.5 minutes. Considering the short execution time of each job, and the overheads in scheduling and running Grid jobs, this is an excellent result and compares favourably with a theoretical minimum parallel execution time of 5.3 minutes. Figure 8 shows the execution profile of the run – peaking at 517 parallel jobs. Some of the overheads are visible as a ramp up and ramp down of jobs at the beginning and end of the run.

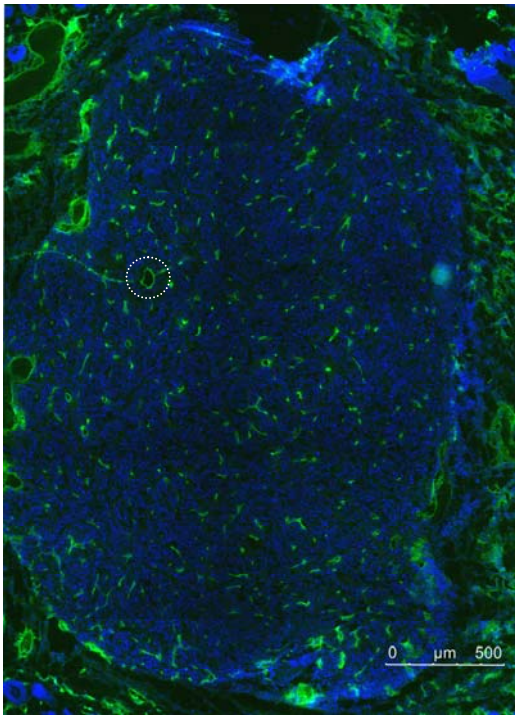


Figure 3 – Bird's-eye view of tumour: cell nuclei in blue (Hoechst stain) and the blood vessels are green (RCA-lectin-FITC). A particular area of interest is marked with a white circle for high- resolution confocal scanning.

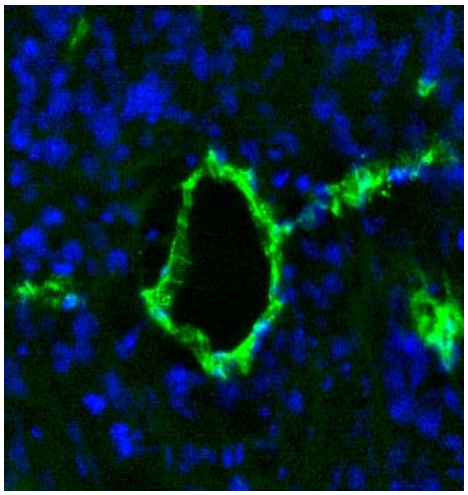


Figure 4 – Magnified area of interest.

The result allows the operator to quickly determine the optimal threshold values, and moreover, see the effect of different values on the resulting image. By examining the movies, it is clear that a number of images contained very little data – this was due to the choice of threshold values for these images. Because all movies are displayed simultaneously, we are able to discard the over-filtered movies very quickly and focus on the ones that produce the best results.

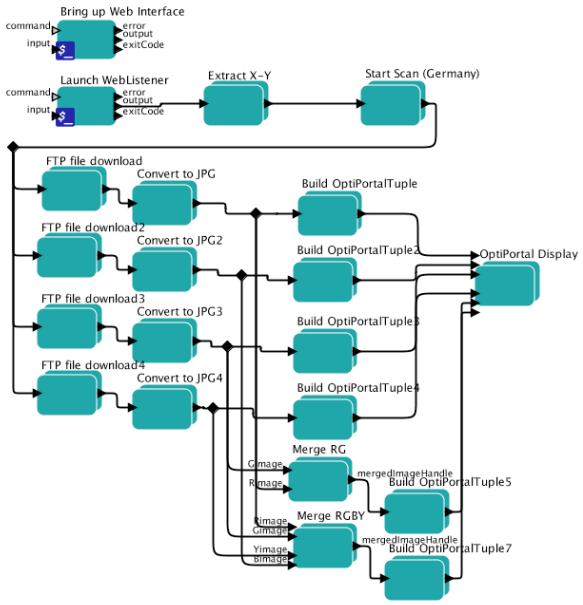


Figure 5 – Workflow 2 – Zooming into area of interest.

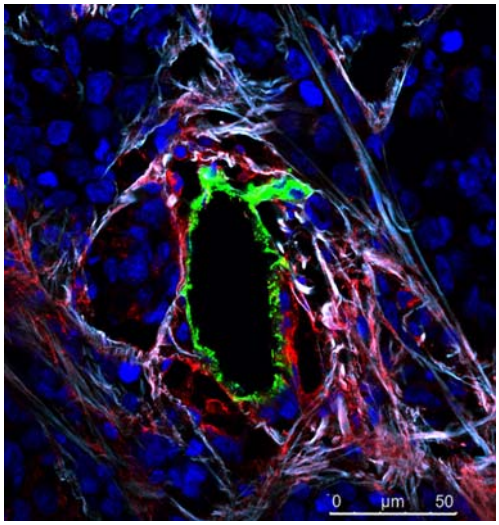


Figure 6 – Output of higher resolution confocal scan. Cell nuclei are blue (Hoechst stain), blood vessels are green (RCA-lectin-FITC), tumour targeting antibody is in red, and connective tissue (anti-tenascin antibody) is in cyan.

V. Conclusions & Further Work

The experiment discussed in this paper, and the software we have developed to support it, demonstrate that a new era of experimental science has begun. Users no longer need to be in the same room as their instruments; we have demonstrated that it is possible to control instruments on the other side of the world, yet process, analyse and store the data they produce locally. Moreover, we demonstrate that scientific workflow tools can provide a robust mechanism for specifying and executing experiments. In order to support this, we used our modifications to Kepler that allow it to execute workflow components in parallel on the Grid (Nimrod/K), and we developed new actors that provided missing functionality.

The work to date is only the beginning. While they are powerful, existing workflows contain too many low

level details, making them difficult for non-specialists to use. For example, users must specify operations that copy data across Grid nodes, and this is both complex to create, and difficult to modify. Moreover, complex (but more efficient) transfer patterns, such as third party transfers, are very difficult to specify. Workflows also contain too many domain specific parameters that make them difficult to reuse and share. For example, file system locations, and URLs are typically coded in the workflow. Accordingly, we are developing a data transport layer for Nimrod/K that abstracts the way data is moved around the Grid.

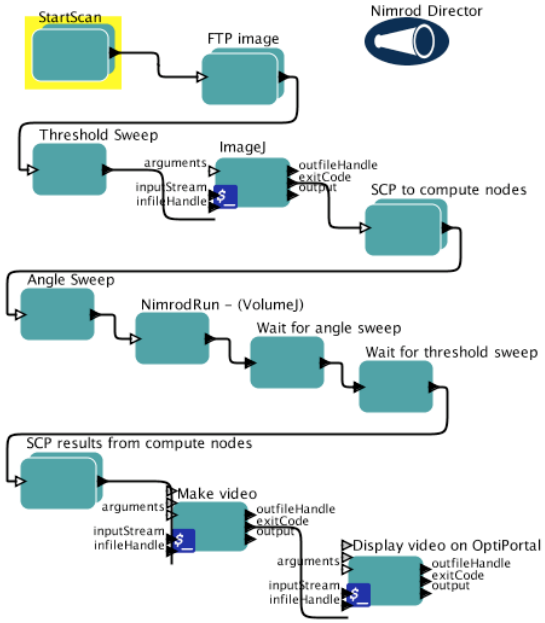


Figure 7 – Workflow 3 – Image cleanup and rendering

Our current implementation requires many enhancements before it can be used routinely. For example, we need to enhance the functionality in libraries like CGLX and SAGE in order to create a more flexible working environment. Whilst the OptiPortal Display actor has demonstrated that it is possible to link a workflow engine with the imaging environment, this link needs to support more data types and support a more interactive mode of operation – currently the libraries are biased towards the separate production of images and their display.

The Web Service APIs we have specified to the Leica microscopes are simple, and require enhancement to enable more control of the instruments. At present, they mimic the manual mode of operation, but we need to expose the significant complexity of the basic instruments by a powerful, but safe, set of APIs. We will further develop these over the coming months.

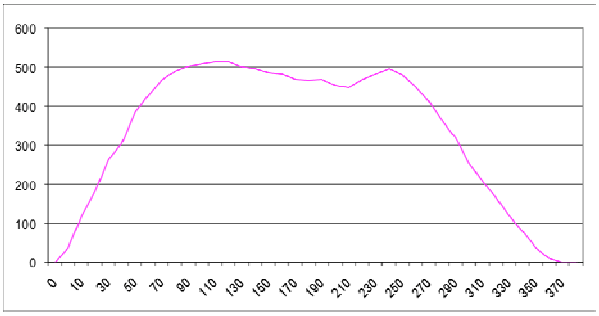


Figure 8 – Execution profile for rendering jobs

ACKNOWLEDGEMENTS

The Nimrod/K project is supported by an Australian Research Council (ARC) Discovery grant. The collaboration between Monash University and Leica Microsystems is supported by an ARC Linkage grant. Hoang Nguyen was supported under Monash Undergraduate Research Projects Abroad (MURPA) funding.

REFERENCES

[1] Abramson, D. Giddy, J. and Kotler, L. "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid," In *Int'l. Parallel and Distributed Processing Symposium (IPDPS),* Cancun, Mexico, May 2000.

[2] Abramson, D., Enticott, C and Altinas, I. "Nimrod/K: Towards Massively Parallel Dynamic Grid Workflows", IEEE Supercomputing 2008, Austin, Texas, November 2008.

[3] De Roure, D., Goble, C. and Stevens, R. (2009) "The Design and Realisation of the myExperiment Virtual Research Environment for Social Sharing of Workflows". Future Generation Computer Systems 25, pp. 561-567. doi:10.1016/j.future.2008.06.010.

[4] DeFanti, T. et al. (2009) The OptIPortal, a scalable visualization, storage, and computing interface device for the OptiPuter", Future Generation Computer Systems, Volume 25, Issue 2, February 2009, Pages 114-123, ISSN 0167-739X, DOI: 10.1016/j.future.2008.06.016.

[5] Görtler, J. Berghoff, M. Kayser, G. and Kayser, K. "Grid technology in tissue-based diagnosis: fundamentals and potential developments". Diagnostic Pathology, 2006.

[6] http://bij.isi.uu.nl/vr.htm

[7] http://kepler-project.org

[8] http://rsb.info.nih.gov/ij/

[9] http://vis.ucsd.edu/~cglx/main.html

[10] http://www.evl.uic.edu/cavern/sage/index.php

[11] http://www.messagelab.monash.edu.au/nimrod.

[12] Jain, R. K. (2005) Normalization of tumor vasculature: an emerging concept in antiangiogenic therapy. Science 307: 58-62.

[13] Jain, R.K., Munn, L.L., Fukumura, D., 2002, Nature Rev. Cancer 2, 266-276

[14] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., Tao J. and Zhao, Y. "Scientific Workflow Management and the Kepler System", *Concurrency and Computation: Practice & Experience*, Special Issue on Scientific Workflows, 2005.

[15] Lundin M, Lundin J, Isola J. "Virtual Microscopy: applications in diagnostic pathology", Journal of Clinical Pathology. 2004;57:1250-1251.

[16] Oinn, T. Addis, M. Ferris, J. Marvin, D. Senger, M. Greenwood, M. Carver T. and Glover, K. Pocock, M.R. Wipat, A. and Li. P. "Taverna: a tool for the composition

and enactment of bioinformatics workflows", Bioinformatics, 20(17):3045-3054, Oxford University Press, London, UK, 2004.

[17] Pan, T. Jewel, S. Catalyurek, U. Wenzel, P. Leone, G. Hastings S., Oster, S. Langella, S. Kurc, T. Saltz, J. and Cowden, D. "Virtual Microscopy: Distributed Image Storage, Retrieval, Analysis, and Visualization", Parallel Computing for Bioinformatics and Computational Biology, 2006.

[18] Polyak, K, Weinberg, R.A. (2009) Transitions between epithelial and mesenchymal states: acquisition of malignant and stem cell traits. Nat Rev Cancer 9: 265-273.

[19] Psaila, B., Lyden, D. (2009) The metastatic niche: adapting the foreign soil. Nat Rev Cancer 9: 285-293.

[20] Tan, J., Abramson, D. & Enticott, C. "REMUS: A Rerouting and Multiplexing System for Grid Connectivity Across Firewalls", Journal of Grid Computing, Online version doi:10.1007/s10723-008-9104-1, June 2008.

[21] Yang, L. Chen, W. Meer, P. Salaru, G. Goodell, L. A. Berstis, V. and Foran D. J., "Virtual Microscopy and Grid-enabled Decision Support for Large Scale Analysis of Imaged Pathology Specimens". IEEE Transactions on Information Technology in Biomedicine, 2009.

[22] Yu, J. and Buyya, R. "A Taxonomy of Workflow Management Systems for Grid Computing," *Technical Report GRIDS-TR-2005-1,* Grid Computing and Distributed Systems Laboratory, Univ of Melbourne, 2005. http://www.gridbus.org/ reports/ GridWorkflowTaxonomy.pdf.

[23] Yuste, R., 2005, Nature Methods 2, 902-4.

Robust Workflows for Science and Engineering

§ David Abramson, Blair Bethwaite, Colin Enticott, Slavisa Garic, Tom Peachey
† Anushka Michailova, Saleh Amirriazi, Ramya Chitters

§ Faculty of Information Technology,
Monash University,
Clayton, 3800, Victoria, Australia

† Department of Bioengineering,
University of California San Diego, and San Diego Supercomputer Centre
9500 Gilman Drive, La Jolla, USA

ABSTRACT

Scientific workflow tools allow users to specify complex computational experiments and provide a good framework for robust science and engineering. Workflows consist of pipelines of tasks that can be used to explore the behaviour of some system, involving computations that are either performed locally or on remote computers. Robust scientific methods require the exploration of the parameter space of a system (some of which can be run in parallel on distributed resources), and may involve complete state space exploration, experimental design or numerical optimization techniques. Whilst workflow engines provide an overall framework, they have not been developed with these concepts in mind, and in general, don't provide the necessary components to implement robust workflows.

In this paper we discuss Nimrod/K - a set of add in components and a new run time machine for a general workflow engine, Kepler. Nimrod/K provides an execution architecture based on the tagged dataflow concepts developed in 1980's for highly parallel machines. This is embodied in a new Kepler 'Director' that orchestrates the execution on clusters, Grids and Clouds using many-task computing. Nimrod/K also provides a set of 'Actors' that facilitate the various modes of parameter exploration discussed above. We demonstrate the power of Nimrod/K to solve real problems in cardiac science.

1 INTRODUCTION

Scientific workflows have emerged as a powerful technique for specifying and executing complex in-silico experiments. Whilst there are many engines available, almost all share the ability to script data manipulation and computational pipelines that invoke a series of steps, often utilizing distributed resources. Pipeline steps can range from those that perform simple computations locally, to accessing external instruments and databases, through to significant simulations on remote parallel machines. A good review of workflow engines can be found in [29].

At the same time, in-silico design has become a common practice in research and industry. Computer simulations are routinely used in engineering design allowing for complex structures to be simulated with significant accuracy. For example, fluid flow, crack propagation and electronic simulations are routinely used in product design. Likewise in research, robust methods make it possible explore many different scenarios fairly cheaply. In spite of this, software is typically developed with particular packages in mind, and is often limited to a single domain. For example, mechanical design software might include powerful optimization procedures, but these are embedded in the packages and are optimized for particular engineering calculations such as stress analysis or crack propagation. Or, a drug docking code might contain routines that explore different drug configurations, but these are tailored to the chemistry of drug-protein interactions. As a result, users can only apply robust search methods if they are embedded in the applications of interest.

Most workflow engines, on the other hand, are designed for generalised application. They include components (which we will refer to as *actors* in this paper) that perform a wide range of activities. Such actors might form part of the dataflow of a workflow, and are used to generate, manipulate and transform data. General invocation actors can execute arbitrary software tools, making it possible to leverage existing packages. Other actors might be responsible for control flow, and route data to different actors depending on the state of the computation. However, apart from a few projects [28], workflows have not been widely used for robust design. This is partly because they are missing actors that implement design templates.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MTAGS'09, Nov 16, 2009, Portland, OR, USA.
Copyright © 2009 ACM 978-1-60558-714-1/09/11... $10.00.

In this paper we discuss the development of such actors, and consider a range of different design techniques – from complete enumeration to targeted search. Importantly, all of these techniques exhibit significant parallelism and can exploit highly parallel platforms by running different scenarios in parallel. We have implemented a number of new actors in an existing workflow system called Kepler [6][7][14][18][19]. When coupled with a parallel execution mechanism called TDA, we can explore multiple designs using clusters, Grids and Clouds. We demonstrate the power of a new tool called Nimrod/K to solve real problems in cardiac science.

2 ROBUST DESIGN

2.1 Techniques

Robust design involves considering more than a single data point in analysing a system. Whilst this seems like an obvious statement, it is remarkable how many scientific experiments (especially in-silico ones) only consider a single, or small number of data points. This may occur because the user may simply assume that the system is well behaved, or because any more rigorous exploration may be prohibitively expensive or infeasible.

Robust design involves understanding the parameters of a system, and exploring the design space using rigorous methods. At one end of the spectrum is complete enumeration, in which parameter ranges are broken into discrete steps, and the cross product of values is generated and explored. Complete enumeration often generates too many designs, and thus there is interest in techniques that reduce the number of scenarios evaluated in a controlled way. One of these, Fractional Factorial Design (FFD) [22], solves this by re-expressing an experimental design in terms of each of the parameters and combinations of those parameters. For example, it assumes an expression in which a set of primary terms are calculated solely on single parameter values, followed by secondary terms that concern all combinations of two parameters, tertiary terms that concern all combinations of three parameters, etc. Using this approach, it is possible to increase the resolution of the model until all combinations of all parameters are considered – this is the equivalent of complete enumeration. FFD assumes that higher order combinations have little effect on the output of a model, and thus these can be removed. The technique uses a variety of visualizations that clearly show the effects of the various parameters, and this makes it possible to prune the search space in a controlled way.

Both complete enumeration and FFD generate the scenarios statically – that is, they decompose the parameter values (and ranges) into discrete instances, and send these off for evaluation as a separate phase. When used for in-silico design, such a technique works well with commodity schedulers such as PBS, Condor, etc, where the jobs can be placed in a queue for execution. An alternative to generating all scenarios up front is to use an iterative algorithm that evaluates the quality of solutions before choosing a new set of scenarios. This mode of operation is well suited to automatic optimization, in which the goal is to minimise or maximise some objective cost function. Various optimization algorithms can be employed – ranging from heuristics that understand the design space, through to generic algorithms such as Genetic Algorithms. Optimization of this type opens up a new range of problems called inverse problems, in which the parameter values are computed to minimise the difference between some model output and a known solution. Inverse problems are common, and are also difficult to solve in general. Accordingly, it is desirable to have a range of search algorithms available.

2.2 Nimrod's approach to robust design

The Nimrod tool family automates the techniques discussed in the previous section using many-task computing (MTC) [25]. It targets computational models that are time consuming, and executes these on a range of platforms from high-end workstations to large compute clusters Nimrod consists of four related tools:

- Nimrod/G, which performs complete enumeration [5];
- Nimrod/E, which performs fractional factorial design [22];
- Nimrod/O, which performs optimization [1]; and
- Nimrod/K, which executes workflows [4].

Nimrod/G, E and O are configured by a small declarative description called a *plan* file. A plan file describes the parameters of interest, their types, values and ranges, and also a set of tasks that tell the system how to execute the model. Figure 1 shows the relationship between these three tools, and how they are exposed to users by a Web portal.

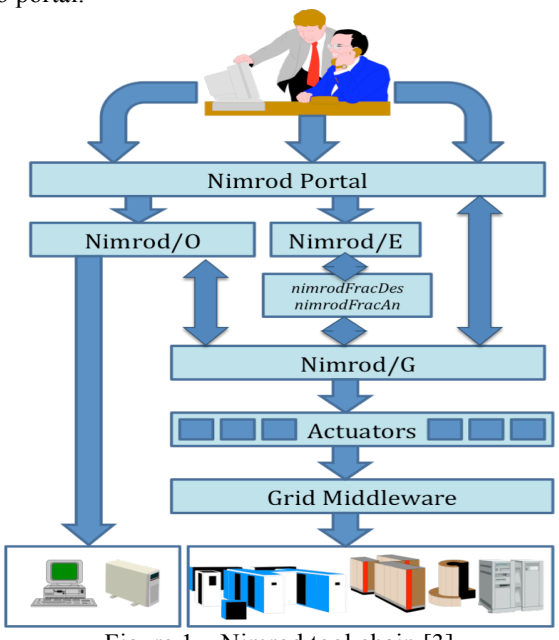


Figure 1 – Nimrod tool chain [3]

As well as functioning as a user level tool, Nimrod/G contains an API that allows other applications to schedule and execute jobs. This API, used by Nimrod/E and Nimrod/O, makes it possible to evaluate models on demand. Whilst Nimrod/E generates a set of jobs statically, Nimrod/O generates them on the fly using a range of non-linear optimization algorithms. After each batch of jobs is executed, the objective cost function values are returned to Nimrod/O, which then decides on the next set to explore. All tools exploit parallel and distributed resources by leveraging a range of Grid middleware – the most mature and commonly available interface is based on the Globus Toolkit [15][16].

Nimrod/G and Nimrod/E generate as many jobs as possible based on the resolution of the parameter specifications. Nimrod/O generates jobs in parallel batches, using parallel versions of various search algorithms where possible [1][23][24]. Nimrod/O also executes multiple searches concurrently in order to handle non-linear problems that have multiple local minima. As a consequence, all three tools generate multiple tasks. A variety of schedulers allocate jobs to resources, ranging from a first-come-first-serve heuristic that simply allocates jobs to the next available resource, through to sophisticated schedulers based on a computational economy [11].

Nimrod/K is a new version of the tool family, and is discussed in more detail in Section 3.

3 THE NIMROD/K WORKFLOW ENGINE

Nimrod/K combines scientific workflows with the robust design techniques discussed in the previous section. It builds on the Kepler workflow engine [4], which itself is built on top of the Ptolemy environment [18]. Kepler inherits a wide range of *orchestration* mechanisms from Ptolemy, enabling different programming models ranging from static dataflow to discrete event simulation. Kepler also includes an enormous range of pre-defined actors – these components can be extracted from a library and placed in workflows. Actors execute a range of functions. For example, some actors manipulate the contents of files; some invoke computations; whilst others perform data manipulation. Kepler actors are written in Java, and thus new actors can be created, filed in libraries, and invoked in new workflows. For a more complete description of the Kepler environment see [6][7][14][18][19]. Figure 2 shows a workflow as envisaged at design time, showing three actors connected by arcs. Tokens move between the actors conveying data, but the execution semantics are described by the choice of director.



Figure 2 – Static workflow of 3 actors

3.1 The Tagged Dataflow Director (TDA)

As discussed, Ptolemy directors control the way a workflow is executed. Kahn Process Networks (PN) and Synchronous Data Flow (SDF) are two of the more common directors used in scientific workflows. Both of these execute actors when data, passed as tokens, is available on actors' inputs. SDF calculates a schedule statically, whereas PN supports bounded queues of tokens and leaves actors in a ready state until tokens arrive. While powerful, neither of these directors exploit parallel platforms in a way which supports parameter sweeps and search. For example, SDF only fires one actor at a time, regardless of how many tokens are active in a workflow. Thus, even when multiple tokens are queued, the workflow cannot exploit this concurrency. PN, which does allow some inter-actor concurrency, only runs one copy of each actor at a time. When used to perform robust design, however, we want to generate a large number of scenarios and execute these in parallel. Accordingly, we want the number of instances of any given actor to increase dynamically as more data tokens are sent to it.

To solve this problem, we created a new director called TDA. TDA implements a Tagged-Dataflow Architecture originally designed for the MIT, RMIT and Manchester data flow machines [17][2][9]. TDA augments data tokens with a tag, or colour field. When multiple tokens of different colours are available on the inputs of an actor, multiple independent instances of the actor are invoked. Thus, by controlling the colouring and de-colouring of token, the workflow can scale to use an arbitrary number of processors. Figure 3 shows the flow of tokens in this workflow when only one token is present on the arcs – 3(a) shows a token on the input of actor 2, and 3(b) shows the token emitted by actor 2 after it has executed. Figure 4 shows the flow of multiple coloured tokens on arcs – 4(a) shows 3 tokens on the input to actor 2, and 4(b) shows the multiple copies of that actor running in parallel.

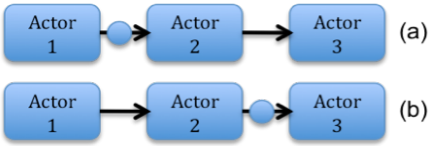


Figure 3 – Tokens in static workflow

The TDA implementation leverages a feature of Ptolemy that "clones" actors. The TDA maintained a set of queues for each actor. When multiple tokens of different colour arrive on an actor's input, TDA creates new copies of the actor using cloning. These clones are destroyed after execution, which allows the graph to dynamically expand and contract as the concurrency changes. More details are discussed in [4].

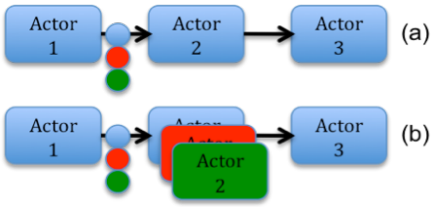


Figure 4 – Tokens in dynamic workflow

3.2 Nimrod/K Actors

Three new families of actors have been created to support the robust design methods discussed in Section 2.

3.2.1 Nimrod/G Actors

As discussed, Nimrod/G takes a cross product parameter values and computes a set of scenarios. Parameter ranges and values are normally provided in a plan file, although they can be added dynamically through the Nimrod/G API. In Nimrod/K, we have created a new actor that takes the same parameter descriptions used in plan files, and it emits a series of tokens – one per instance.

Figure 5 shows the way this actor is instantiated in a workflow, and also shows the configuration of the parameter values. In this case, ParameterSweep is linked to a MatlabExpression actor that executes a Matlab program. The tokens that flow into the Matlab actor contain actual values for the various parameters – in this case 9 different variables. This actor works with the current Ptolemy directors, however, when used in combination with the TDA, parallel execution of the different parameter combinations allows the workflow to execute efficiently on the Grid.

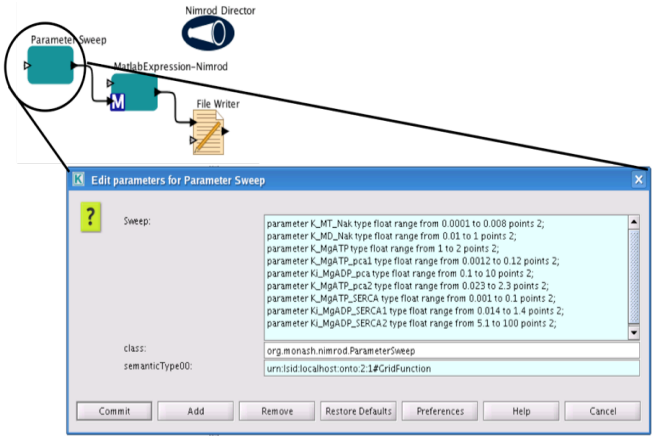


Figure 5 – Nimrod/G actors and configuration control

3.2.2 Nimrod/E Actors

Nimrod/E is similar to Nimrod/G in that it generates a set of experiments. However, it uses the Fractional Factorial Design algorithm to decide which combinations to explore. These are produced by the ExperimentalDesign actor. The plan file syntax for Nimrod/E is a superset of Nimrod/G, and contains additional statements that control the resolution of the model. In addition to the ExperimentalDesign actor, an EffectsEstimator actor computes the effects of the various parameter combinations. These are streamed to two different graphing actors, one that generates a Daniel plot, and another that generates a Lenth plot. These two diagrams show the effects of various parameter combinations in a graphical form. Lenth plots show the effects of various parameter combinations. Parameter combinations, which are shown as individual dots, are plotted using the Y axis range to indicate their contribution. Thus, dots that lie close to the X axis have little effect on the output. In the Daniel plots, the results are displayed using a cummulative graphing technique. Here, insignificant combinations cluster on a straight line, and significant ones appear off the line. Using these two forms of display, it is very easy to spot the significant combinations. Figure 6 shows a workflow that uses all these actors. Figure 7 shows examples of Daniel and Lenth plots.

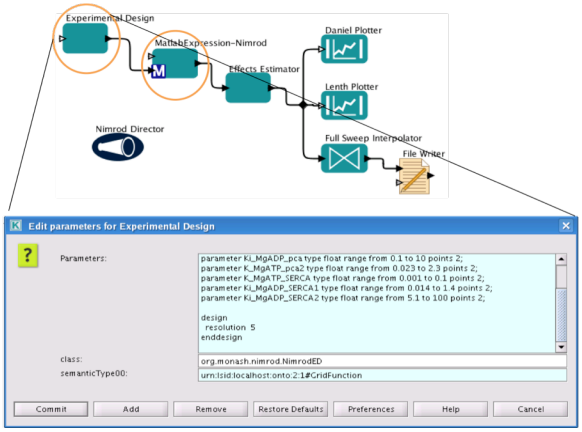


Figure 6 – Experimental Design actors

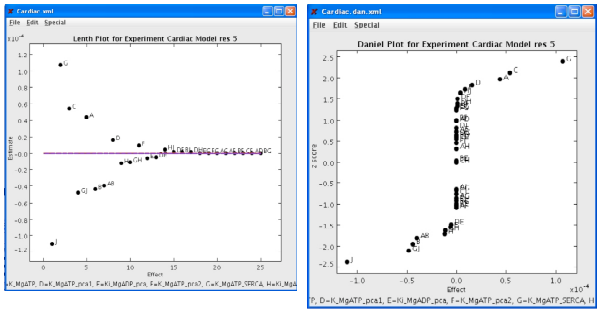


Figure 7 (a) Lenth Plot (b) Daniel Plot

3.2.3 Nimrod/O Actors

Optimization algorithms combine well with workflows because they usually involving repetitive looping in which results are passed from one iteration to the next. Implementing Nimrod/O functionality within Kepler

exposes the components of the optimization process, giving the user a clear view of the data flows and facilitating substitution of these components. Accordingly, we have built a set of actors that support the generic actors of an optimization algorithm, and also specific actors that implement specific algorithms.

Figure 8 shows a generic optimization algorithm and the actors that support it. First, the domain of the search is defined by specification of the input parameters using a similar method to Nimrod/G plan file syntax. Within this domain the next component will select starting points for the multiple optimizations. Each starting point will begin an optimization run using the specific optimizer actor. This actor generates sets of points that are sent for evaluation, a "batch of jobs" in Nimrod/O's terminology. Running the models represented by these jobs produces numerical results that are used to decide the next generation of points. This cycle continues until some convergence criterion has been achieved whereupon a final point is forwarded as the result of the search. Figure 8 also shows a component that evaluates constraint conditions imposed by the user. This calculates the margin by which a search point violates any constraints, and imposes penalties accordingly. If the constraint is a "hard" one then the point is completely forbidden, obviating the need for model evaluation.

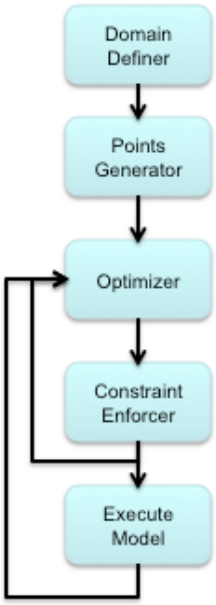


Figure 8 – Generic optimization actors

We are currently building actors for the existing Nimrod/O optimization methods, namely gradient descent, simplex, genetic algorithms and simulated annealing.

4 CASE STUDIES: CARCIAC IONIC-METABOLIC MODEL

In this section we illustrate the three design techniques in Nimrod/K using a real world case study in cardiac modelling. The electrical activity of the heart is based on cycles of ion transfer via cell surface membrane. There has been much research on developing robust and accurate models of myocyte behaviour so that in-silico experiments can be performed on complete cardiac systems. The research has significance from basic science through to therapeutic drug design.

Cardiac excitation-contraction coupling is a sequence of well-orchestrated events. Both excitation and contraction processes and their interactions are required for an integrative model of cardiac electromechanical interactions [26]. A number of mathematical models have been developed to study excitation-contraction coupling in ventricular cardiac cells. Recently, two of the authors of this paper integrated additional ionic-metabolic equations [8][20] into the Shannon et al. four compartment cell model [26]. The updated ionic model is more complex due to the multiple domains and increased number of unknown model parameters, so it implies a less stable set of ordinary differential equations systems. For this reason, any information gained based on this model can yield a significant level of misunderstanding if one wants to use it investigate the excitation-contraction coupling in ventricular myocytes.

As a result of this integration, we are interested in two different experiments. First, we want to calibrate the new model so that it matches real physiological data. Second, we want to explore the sensitivity of the model to the parameters, and determine whether there are some combinations that are more important than others. Accordingly, the first experiment is performed with Nimrod/OK and the second with Nimrod/EK.

4.1 Nimrod/OK Experiment

After adding the new channel and metabolic factors for both normal condition and ischemia, we confirm the results against physiological data. We then stabilize the model for a heart rate of 0.5Hz and a longer performance duration (3 min). In both, the focus is on calcium transient concentrations (dyad, sub-membrane space, cytosol, sarcoplasmic reticulum), and action potential shape and duration, to determine the stability and accuracy of our model. In order to validate the model, some final outputs are compared against experimentally known data [20][26]. This involved exploring a range of numbers for nine input metabolic constants and selecting the combination that produces the closest output to the desired value.

This experiment performed simplex searches over the nine-dimensional space, using the workflow shown by Figure 9. The computations are performed on a distributed Grid of Linux clusters, as shown in Table 1.

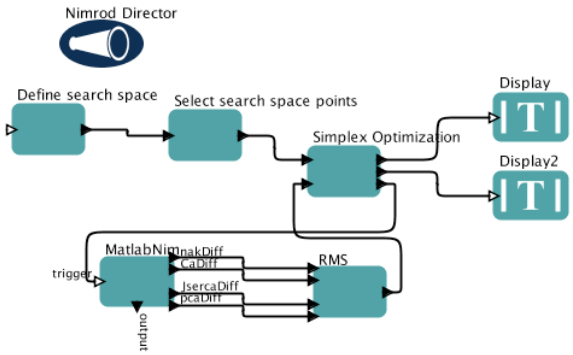


Figure 9 – Optimization run in Nimrod/OK.

Machine	# Cores	Hardware	Location
East	160	Intel(R) Xeon(R) E5310 @ 1.60GHz	Monash University, Melbourne
	64	Intel(R) Xeon(R) 5160 @ 3.00GHz	
West	160	Intel(R) Xeon(R) E5310 @ 1.60GHz	Deakin University, Geelong
South	160	Intel(R) Xeon(R) E5310 @ 1.60GHz	RMIT, Melbourne

Table 1 – Properties of Grid Testbed

128 starting points are randomly selected within the domain. Table 2 summarises the results of these searches, showing the best, worst and average search results. We also show the number of jobs and batches required to achieve the best and worst results, and the average number of jobs and batches. The best search gives a considerably better (smaller) result than the average, at the expense of more computational effort as shown by the numbers of jobs and batches of jobs. This shows the utility of multiple searches.

Job executions averaged 3m 21s, but with substantial parallelism the full experiment was completed in 7h 37m. Figure 10 plots the number of jobs executing over the duration of the experiment. Initially the simplex method evaluates the objective at the vertices of the starting simplex. In this case each of the simplexes has ten such vertices, so the experiment began with 1280 jobs. Thereafter, most iterations require just four new evaluations, so the load dropped to 512 jobs, except for the occasional peak where a new simplex was required. As searches completed the load dropped in stages; the final longest running optimization required only four processors but dominated the experiment wall clock time.

Index	Objective	Jobs	Batches
Min	0.00042	154	37
Average	0.0015	142	35
Worst	0.0038	34	7

Table 2 – results for searches over the full domain

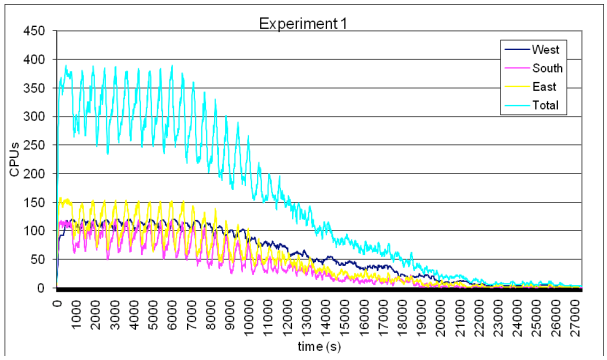


Figure 10 – job concurrency for full domain search

4.2 Nimrod/EK Experiment

The main goal of this experiment was to examine the changes in which total ionic flux(s) in the compartment of interest may play the largest role in determining specific model outputs (action potential (AP) and Ca transients in the four cell sub-domains) in ventricular cardiac cells isolated from the sub-endocardium tissue layer in rabbits. We use Nimrod/EK to perform a parameter sweep and measure the effects of each of the parameter combinations, as summarised by the workflow in Figure 11.

In agreement with experiment [26], results suggest that small changes in L-type calcium and sodium/calcium exchanger total fluxes in the junctional cleft may significantly affect the cell function. Specifically, Figure 12 shows a Lenth plot for the AP duration. Here, there are few combinations that actually exhibit any significant effect because most of them are clustered around the X axis. This is reenforced in the Daniel plot in Figure 13, in which significant combinations appear off the line, and those on the line are insignificant. The same effects are apparent in the cytosolic Ca-peak in Figures 14 and 15. Surprisingly, the studies suggested that small changes in the junctional chloride/calcium flux also may have prominent effect on the model outputs. New experiments need to be perfomed to test this hypothesis. The fact that both the action potential duration and cytosolic calcium peak are impacted by the same fluxes distributions strengthen further our findings.

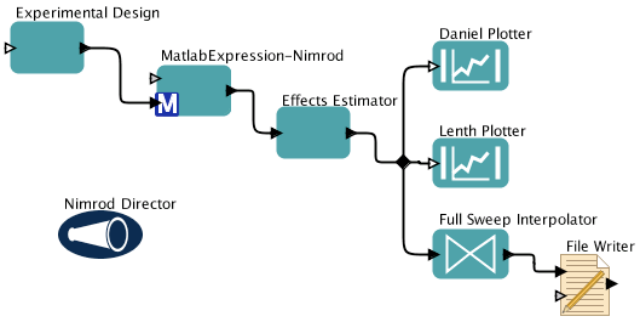


Figure 11 – workflow using Nimrod/EK

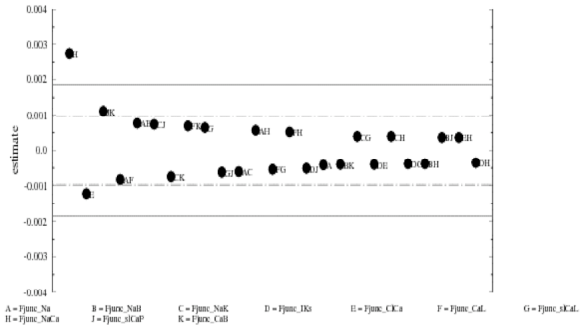


Fig. 12 Length plot for AP duration

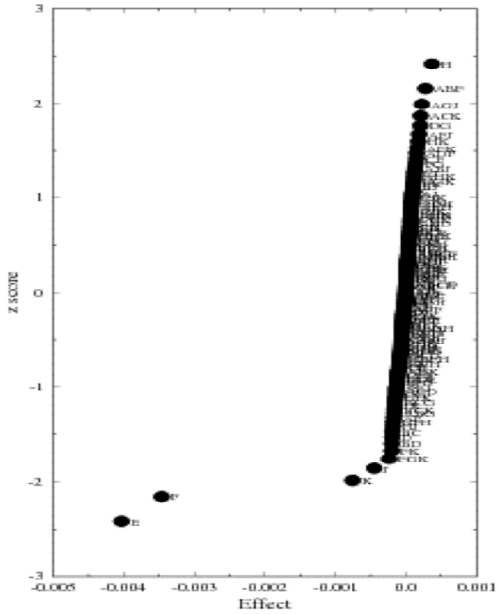


Fig. 13 Daniel plot for AP duration comparison

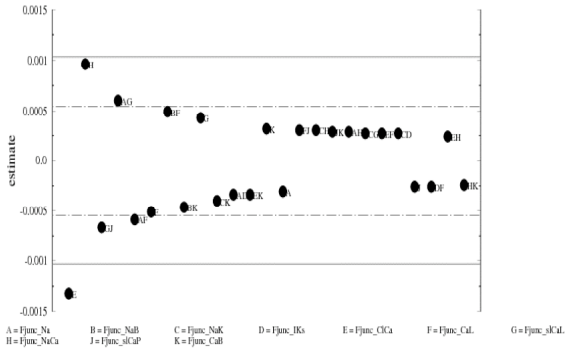


Fig. 14 Length plot for cytosolic Ca-peak comparison

5 RELATED WORK

While most workflow engines support some form of Multi-task Computing, many of them are remarkably static – that is, the graphs that are generated do not typically change size at run time and thus the level of parallelism cannot be varied easily. This static code generation strategy means there are usually only two ways to specify parallel activity. Either,

1. build a very large static DAG with cloned sub-graphs for each parallel activity; or
2. write a workflow that contains logic to spawn parallel activities.

The first of these means that the amount of parallelism cannot change once the graph has started execution, which is restrictive. Further, if there is a lot of parallelism, the graph can become very large and difficult to manage. For example, a workflow that wishes to process the contents of a database, in parallel, must be replicated many times to allow them to run in parallel. The underlying graph can become very large, but it also contains a high degree of redundancy since the same operations are applied to each database element. This is the approach taken with tools like APST [12] and Pegasus/DAGMAN [13]. The second alternative supports dynamic parallelism, but pushes the onus onto the programmer to add the task management logic. For example, users must explicitly code for parallel execution, and incorporate loops that process the contents of the database, provide mechanisms that spawn independent calculations and then synchronize the results. This process significantly complicates the task of writing a workflow. In spite of this, the approach is typical of systems such as Kepler, for example.

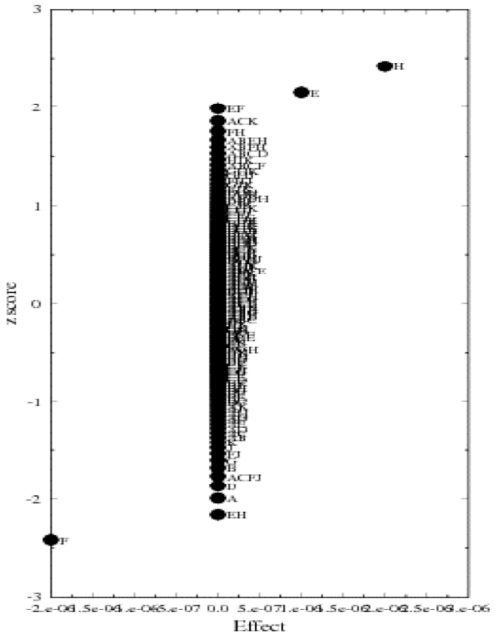


Fig. 15 Daniel plot for cytosolic Ca-peak comparison

An alternative to static code generation is to support the dynamic spawning of sub-graphs. This is the approach effectively taken in Nimrod/K, Swift [30] and the most recent additions to Taverna [21]. These systems are simpler to use, because the task management code is embedded in a

middleware layer, and they can scale from small to large amounts of parallelism very quickly and dynamically.

Almost no workflow engines support the search optimization algorithms implemented in Nimrod/O, with the notable exception of Geodise [28]. Geodise's goal is to "expose optimisation services in a flexible, generic interface that can be easily integrated into various environments and used to compose different optimisation workflows." [28]. In Geodise, optimization algorithms are exposed as Web services, which can be invoked by a workflow system, or any other programming language that can execute Web service calls. In our case we have embedded our work in an existing workflow engine, Kepler, and have built Kepler actors that perform the optimization functions.

6 CONCLUSIONS

This paper has provided a historical view of the Nimrod tool family, and more importantly, showed how we have leveraged an existing workflow engine to incorporate Nimrod's advanced search techniques. Specifically, we have added complete enumeration, fractional factorial design and optimization methods to Kepler by building special actors. In order to support parallel execution, we have incorporated a new Kepler director called TDA. The result is a very flexible workflow environment that can be used for automatic design. Importantly, we leverage the enormous number of existing actors in Kepler and Ptolemy.

We have demonstrated the applicability of Nimrod/K by various case studies in cardiac science. These have shown that we can solve a complex inverse problem framed as an optimization problem, and then using the Nimrod/OK actors. We have also used Nimrod/EK to explore the role of various parameters in the mathematical model.

Nimrod/K is an active research project. We are currently exploring a number of research themes using this as a base. First, we are building a data transport framework that removes the need for users to explicitly manage data staging and movement in their workflows. When combined with the Nimrod/K engine, this will provide a very powerful distributed computing platform. Second, we are designing and building arrange of scheduling heuristics that work with the data transport and parallel computing models discussed [10][27]. Third, we are building interfaces with powerful display technologies, and these allow a user to be immersed in the optimization process.

ACKNOWLEDGEMENTS

This project is supported by the Australian Research Council under the Discovery grant scheme We acknowledge our colleagues in the MeSsAGE Lab at Monash University and the Cardiac Mechanics Lab at UCSD. We also acknowledge the US NSF funded PRIME project, (supporting Amirriazi and Chitters) and thank colleagues, in particular Peter Arzberger, for their strong support.

REFERENCES

[1] Abramson D, Lewis A, Peachey T, Fletcher, C., "An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics", SuperComputing 2001, Denver, Nov 2001

[2] Abramson, D and Egan, G, "The RMIT Dataflow Computer: A Hybrid Architecture", The Computer Journal, Vol 33, No 3, June 1990, pp 230 - 240.

[3] Abramson, D., Bethwaite, B., Enticott, C., Garic, Slavisa and Peachey, T. "Parameter Space Exploration using Scientific workflows", ICCS 2009, SU Center for Computation & Technology, Baton Rouge, Louisiana, U.S.A, May 25-27, 2009.

[4] Abramson, D., Enticott, C and Altinas, I. "Nimrod/K: Towards Massively Parallel Dynamic Grid Workflows", IEEE Supercomputing 2008, Austin, Texas, November 2008.

[5] Abramson, D., Giddy J., and Kotler, L. "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid," In *Int'l. Parallel and Distributed Processing Symposium (IPDPS),* Cancun, Mexico, May 2000. http://www.csse.monash.edu.au/~davida/nimrod/.

[6] Altintas, I. Berkley, C. Jaeger, E. Jones, M. Ludäscher B. and Mock, S. "Kepler: Towards a Grid-Enabled System for Scientific Workflows," in the *Workflow in Grid Systems Workshop in GGF10* - The 10th Global Grid Forum, Berlin, March 2004.

[7] Altintas, I., Birnbaum, A., Baldridge, K., Sudholt, W., Miller, M., Amoreira, C., Potier Y. and Ludaescher, B. "A Framework for the Design and Reuse of Grid Workflows" *Intl. Workshop on Scientific Applications on Grid Computing* (SAG'04), LNCS 3458, Springer, 2005

[8] Amirriazi S., Chang S., Peachey T., Abramson D. and Michailova A., *Optimizing Cardiac Excitation-Metabolic Model By Using Parallel Grid Computing*, Biophysics 2008, Long Beach, California, February 2008

[9] Arvind, and Nikhil R.S. "Executing a Program on the MIT Tagged-Token Dataflow Architecture", IEEE Transactions on Computers, Vol. 39, No. 3 (1990)

[10] Ayyub, S. and Abramson, D. "GridRod - A Service Oriented Dynamic Runtime Scheduler for Grid Workflows". 21st ACM International Conference on Supercomputing, June 16-20, 2007, Seattle.

[11] Buyya, R. and Abramson, D. "The Nimrod/G Grid Resource Broker for Economic-based Scheduling", in Market-Oriented Grid and Utility Computing (Wiley Series on Parallel and Distributed Computing), Editors Rajkumar Buyya and Kris Bubendorfer, ISBN-13: 978-0470287682, November 2009.

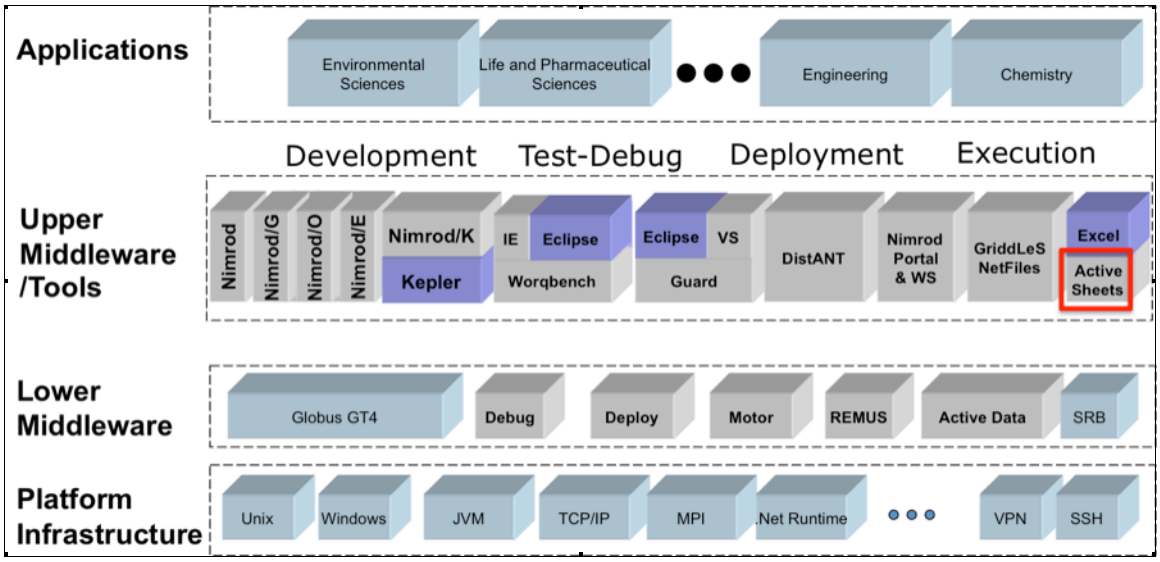
[12] Casanova H. and Berman, F. "Parameter Sweeps on The Grid With APST", chapter 26. Wiley Publisher, Inc., 2002. F. Berman, G. Fox, and T. Hey, editors.

[13] Deelman E., Blythe J., Gil Y., Kesselman C., Mehta G., Patil S., Su M, Vahi K., Livny M., "Pegasus : Mapping Scientific Workflows onto the Grid", Across Grids Conference 2004, Nicosia, Cyprus.

[14] Eker, J., Janneck, J. W., Lee, E. A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y., "Taming Heterogeneity---the Ptolemy Approach," Proceedings of the IEEE, v.91, No. 2, January 2003.

[15] Foster I. and Kesselman, C. “Globus: A Metacomputing Infrastructure Toolkit,” *Int’l J. of Supercomputer Applications*, vol. 11, no. 2, 1997, pp. 115-128.
[16] Foster, I., “Globus Toolkit Version 4: Software for Service-Oriented Systems,” Conference on Network and Parallel Computing, 2005.
[17] Gurd J.R. and Watson I. “Data Driven System for High Speed Parallel Computing (1 & 2) Computer Design, vol.9 nos.6 & 7, June & July 1980, pp. 91-100 & 97-106.
[18] Lee, E. et al, "Overview of the Ptolemy Project," Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March 6, 2001.
[19] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., Tao J. and Zhao, Y. “Scientific Workflow Management and the Kepler System”, *Concurrency and Computation: Practice & Experience*, Special Issue on Scientific Workflows, 2005.
[20] Michailova A., Lorentz W., McCulloch A., *Modelling Transmural Heterogeneity of KATP current in Rabbit Ventricular Myocytes*, AJP-Cell Physiology, 293 (2007), pp C542-C557.
[21] Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M. R., Wipat, A., Li, P., Taverna: A tool for the composition and enactment of bioinformatics workflows. Bioinformatics Journal, 20(17):3045–3054, 2004.
[22] Peachey, T. C., Diamond, N. T., Abramson, D. A. Sudholt, W., Michailova, A. Amirriazi, S., *Fractional Factorial Design for Parameter Sweep Experiments using Nimrod/E,* Journal of Scientific Programming, Volume 16, Numbers 2,3, 2008.
[23] Peachey, T., Abramson, D. and Lewis, A. *Model Optimization and Parameter Estimation with Nimrod/O,* The International Conference on Computational Science, May 28-31, 2006, University of Reading.
[24] Peachey, T., Abramson, D., Lewis, A., Kurniawan, D. and Jones, R. *Optimization using Nimrod/O and its Application to Robust Mechanical Design,* PPAM 2003, Czestochowa, Poland, Lecture Notes in Computer Science, Volume 3019 / 2004, pp. 730 – 737, September 7-10, 2003.
[25] Raicu, I., Foster, I., Zhao. Y. "Many-Task Computing for Grids and Supercomputers", IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08), 2008.
[26] Shannon T. R., Wang, F. Puglisi, J. Weber C. and Bers, D. M. *A Mathematical Treatment of Integrated Ca Dynamics within the Ventricular Myocyte*, Biophysical Journal, 87 (2004), pp 3351-3371.
[27] Smanchat, S., Indrawan, M., Ling, S., Enticott, C. and Abramson, D. “Scheduling Multiple Parameter Sweep Workflow Instances on the Grid”, IEEE e-Science 2009, Dec 9 – 11th, Oxford, UK.
[28] Xue G., Song W., Cox S.J., and Keane A.J., Numerical Optimisation as Grid Services for Engineering Design, Journal of Grid Computing, Vol.2 No.3, 2004, pp223-238.
[29] Yu J., and Buyya, R., A Taxonomy of Workflow Management Systems for Grid Computing”, Journal of Grid Computing, Springer Press, New York, USA.
[30] Zhao, Y., Hategan, M., Clifford, B., Foster, I., von Laszewski, G., Raicu, I., Stef-Praun, T. and Wilde, M. “Swift: Fast, Reliable, Loosely Coupled Parallel Computation”, IEEE Workshop on Scientific Workflows 2007.

Chapter 7: Active Sheets and Spreadsheet Experiments



In this chapter I explore a novel technique that allows users to specify their parametric experiments via spreadsheets. Active Sheets leverages the Grid infrastructure in Nimrod, and allows a spreadsheet cell to reference a computational model that runs on an external Grid resource. (25) discusses the original idea, and (26) shows how this can also support NetSolve computations. (27) is a provisional patent about Active Sheets. This patent was allowed to expire after an initial search failed to locate a partner to commercialize the work.

(1) Abramson, D., Roe, P., Kotler L and Mather, D., "ActiveSheets: Super-Computing with Spreadsheets". 2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference, April 22-26, 2001, pp 110 – 115, Seattle, Washington (USA). 7-1 .. 7-6

(2) Abramson, D., Dongarra, J., Meek, E., Roe, P and Shi, Z., "Simplified Grid Computing through Spreadsheets and NetSolve", Omiya Sonic City, Saitama, Tokyo, Japan, July 20-22, 2004, HPC Asia 2004. 7-7 .. 7-12

(3) Parallel Execution Mechanism for Spreadsheets, Australian Provision Patent PQ8365/01. US Patent lodged (Number 20010056440). 7-13 .. 7-23

Contributions

Roe, from QUT, worked with me on the original Active Sheets idea; Kotler provided a case study in X-Ray physics; Mather provided a case study in computational epidemiology; Dongarra and Meek are colleagues from the University of Tenessee who provided expertise on NetSolve; Shi worked as a research associate with Roe at QUT.

ActiveSheets: Super-Computing with Spreadsheets

David Abramson †, Paul Roe §, Lew Kotler ‡ and Dinelli Mather 8

† School of Computer Science and Software Engineering
Monash University,
CLAYTON, VIC 3168, Australia

§ School of Computer Science and Software Engineering
Queensland University of Technology,
Gardens point Campus
Brisbane QLD 4000, Australia

‡ Australian Radiation Protection and Nuclear Safety Agency
Lower Plenty Road
YALLAMBIE VIC 3085, Australia

8 School of Business Systems
Monash University,
CLAYTON, VIC 3168, Australia

ABSTRACT

Spreadsheets are important business tools. Increasingly they are being used for simulation e.g. to perform risk analysis. Such tasks have far greater computational demands than traditional spreadsheet bookkeeping applications. In this paper we show how spreadsheets can support supercomputing. This is achieved without requiring the spreadsheet user to have specialist tools or knowledge. The key technical innovation is a mechanism enabling the concurrent evaluation of spreadsheet functions. Furthermore the mechanism does not require modification of the standard spreadsheet evaluation engine.

INTRODUCTION

Spreadsheets are popular because they are easy to use and modify, and they support numerical data analysis without programming. A feature of modern spreadsheets is their capability to be extended; in particular custom functions may be incorporated into them, for example complex simulation functions and computational experiments. For these reasons spreadsheets make an ideal front-end for numeric simulations since they support pre and post processing of simulation data without the need for programming.

Since simulations are often computationally intensive they are good candidates for parallel evaluation. Whilst there have been many attempts to accelerate various aspects of simulation programs through parallelisation, an easy way to expose parallelism is to execute the simulation of several different scenarios concurrently. High performance distributed computers provide an excellent platform for this style of execution, because independent simulations can be run on a number of different machines concurrently. We have significant experience with a custom interface that supports this functionality with a research tool called Nimrod [2][1] and its commercial counterpart, EnFuzion[20]. These tools allow us to distribute independent executions of a single simulation in order to perform complex scenarios analysis. They manage the generation of work, distribution of the computation and gathering of the results. The output is very often read back into a spreadsheet for analysis and visualisation once it has been returned from the distributed computers. Spreadsheets allow a range of tools to be used to display results, such as statistical analysis and charting utilities. Consequently, the idea of using a spreadsheet to generate the runs, perform the simulations and gather the results within the one application is attractive because it obviates the user from running multiple independent packages. Complex structures can be established easily, for example, the output of one simulation may be processed in the spreadsheet, and then sent as input to another one. However, for this to work in a spreadsheet setting, parallel evaluation of spreadsheets is required. Unfortunately existing spreadsheets, such as Microsoft Excel [14], have in-built sequential calculation mechanisms. Thus, they are unable to evaluate multiple cells concurrently using the standard execution mechanism in the cell calculation engine.

This paper describes how a conventional spreadsheet may be evaluated in parallel without modifying the core execution engine. The parallel evaluation mechanism is encoded using custom functions, and works in conjunction with the standard built in sequential evaluation mechanism of spreadsheets. The paper begins with a discussion on how to perform high performance simulation on distributed computers. We then discuss how an existing sequential evaluation mechanism can be modified to work in parallel, some implementation issues, followed by a few case studies that illustrate the power of such a system.

HIGH PERFORMANCE SIMULATION

Simulation is widely used in many fields. It allows a user to explore a wide design space without the need to build real world models. Simulation techniques vary from continuous simulations, discrete event simulation and mathematical modelling of real world phenomena using systems of equations (such as partial differential equations). Whilst simulations for simplified models can be quite fast, they are often very slow for accurate simulation using realistic problem sizes. It is not surprising, therefore, that High Performance Computing (HPC) has been used extensively in support of simulation.

A number of different methods have been used in the past to accelerate simulation using parallel machines. For the most part, these consist of parallelising the core algorithms used in the simulation. Because the implementation techniques vary so widely, there is no common method of application. For example, simulations that solve partial differential equations usually use a parallel decomposition of the finite difference representation of the problem [13]. This means that sections of the domain are calculated in parallel, and boundary values are transmitted between the processes as required. Simulations that involve ordinary differential equations may require different solution techniques, and these are, accordingly, parallelised differently. On the other hand, parallel discrete event simulation has received a great deal of attention over the years [21][10][7][19][12][11][18]. Since it is inherently a sequential process, it has been hard to achieve good parallel speedup even though the underlying problem being solved may be highly concurrent. The most successful parallel techniques have involved speculative execution of the event list with subsequent back-off operations when the simulation proceeds incorrectly.

Another technique that has been largely ignored is the area of scenario analysis. In this mode, the same simulation model is run repeatedly using different parameters [5]. Such a technique can be parallelised easily, by simply running the different scenarios concurrently. Whilst the technique is simple, it achieves close to perfect speedup with very little effort. This mode of execution is support by the tools we have developed, namely Nimrod [1][2][3][4] and EnFuzion [20].

When a user describes an experiment to Nimrod and EnFuzion, they develop a *declarative* "plan" file which describes the parameters, their default values, and the commands necessary for performing the work. The system then uses this information to transport the necessary files and schedule the work on the first available machine.

When the user invokes Nimrod on a workstation, the machine becomes known as the "root" machine because it controls the experiment. When the dispatcher executes code on remote platforms, each of these is known as a computational "node". Thus a given experiment can be conducted with one root and multiple nodes, each of a different architecture if required.

However, both Nimrod and EnFuzion expect the user to specify the scenarios they wish to explore using a dedicated user interface. Further, they rely on the user to produce programs that aggregate the results, and if these are to be processed by a spreadsheet tool, they must be imported manually. ActiveSheets is a system that automates this by combining all the functions of Nimrod into the spreadsheet tool itself. Importantly, we rely on an API for EnFuzion that allows it to provide the backend computation platform, whilst the frontend is provided by a commercial spreadsheet tool, in this case, Microsoft Excel.

PARALLEL EVALUATION OF SPREADSHEETS

The mechanism for the parallel evaluation of spreadsheets is based on the dataflow model of computation [6][9]. An expression such as (1+2)*(3+4) may be naturally represented as a graph as shown in Figure 1.

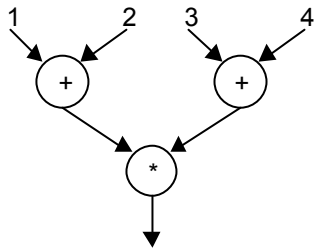


Figure 1 – Dataflow graph exhibiting parallelism

Nodes represent operators; once an operator has all arguments it may be evaluated generating a result which is output. Thus evaluation may be viewed as data flowing along arcs between operators. Operators may evaluate in parallel subject only to arguments being available. This expression can be represented in a spreadsheet as illustrated in Figure 2.

In ordinary spreadsheets cells are always evaluated sequentially. However, if the values in row 1 change we require that cells A2 and C2 are evaluated in parallel, just as the dataflow graph in Figure 1 supports parallel evaluation. Using custom functions, rather than built-in ones like "+", a spreadsheet may be made to behave in this way even though its in built evaluation mechanism is sequential.

	A	B	C	D
1	1	2	3	4
2	=A1+B1		=C1+D1	
3	=A2+C2			

Figure 2 – Spreadsheet representation of Figure 1.

The goal is to evaluate separate functionally independent cells in a spreadsheet in parallel. In particular the goal is to evaluate in parallel the custom functions (e.g. representing a simulation) in such cells. The key to evaluating an otherwise sequential spreadsheet in parallel is to adopt a two stage evaluation process for custom functions. Note that the scheme we are proposing will not work with built-in functions in parallel; however, this is not a serious drawback since most of the time this would not be desirable. In order to trick the otherwise sequential execution mechanism, a custom function sends its arguments together with a representation of the function to a backend computer for parallel evaluation. A table is used to store the current state of the cell, which can be one of the following states: unevaluated, under evaluation or evaluated. The backend may distribute the calculation to any one of a number of machines. The custom function in the spreadsheet returns before evaluation has completed. It returns a *distinguished* error or *undefined* value, which will prevent the cell from being interpreted as a valid value. *By returning immediately without waiting for the custom function's result other custom functions may be evaluated, giving rise to parallel evaluation.* The rest of the spreadsheet is evaluated in the same way driven by the standard built in evaluation[1] mechanism. The parallel evaluation of the spreadsheet is automatic subject only to functional dependencies between spreadsheet cells.

Once results have been calculated on the backend they are sent to the spreadsheet. The results are collected for insertion into the spreadsheet. However the results are not inserted into the spreadsheet directly, rather they are stored in a table so that subsequent re-evaluation of the destination cells will discover them. Re-evaluation of the spreadsheet is forced programmatically. The "second time around" a result is found in the table representing the true value of the function; this is then returned as the function's result.

IMPLEMENTATION ISSUES

The implementation comprises an add-in for Microsoft Excel (version 97 and 2000). Excel is highly customisable through OLE automation; this is used to implement the two phase evaluation strategy which gives rise to parallel evaluation.

Custom functions are implemented in Visual Basic for Applications (VBA) as stubs [15]. These stubs marshal arguments and function identifiers to an ActiveX DLL (COM component). For a given function evaluation, the ActiveX component either sends a request to the back-end machine to evaluate the function or extracts the function's result from a table. In the latter case the result will have been placed there by the backend machine, in response to the first call for the function's evaluation. The key to the implementation is the table that stores the state of custom function evaluation requests. Effectively it serves as a huge cache. Currently the user decides on which custom functions are to be evaluated in parallel on the backend machine.

The implementation supports both inter-cell parallelism and intra-cell parallelism. For example consider a formula such as "=foo(A1)+bar(B1)" where "bar" and "foo" are custom functions, the system will support the parallel evaluation of "foo(A1)" and "bar(B1)".

All backend aspects of the system have been factored into an ActiveX control known as the "adaptor". This enables the system to utilise different backends by simply selecting different adapters. Currently we use a manual builder tool to build the stub code, however this is dependent on the backend. Where metadata is available, the stub code can be automatically generated.

An important goal, which has been realised, is that the system coexists with Excel without requiring any modifications or restrictions to its use. The system may be incorporated into an existing spreadsheet without requiring the spreadsheet to be rewritten – all that is required is for the parallel functions to be available in the backend machine.

CASE STUDIES

In this section we examine two case studies. These are problems that have been solved previously using a combination of EnFuzion and spreadsheets. Whilst we have not re-run the experiments using ActiveSheets, the studies highlight the power of using a single spreadsheet tool for all of the tasks.

Ionisation Chamber Simulation

The Australian Radiation Protection and Nuclear Safety Agency (ARPANSA) provides the Primary Australian Standards for certain Radiological quantities such as Air Kerma, which essentially quantifies photon energy deposition in terms of ionization in air. The calibration of medical equipment used for diagnosis and/or treatment

[1] The terms evaluation and calculation are used interchangeably

against these standards ensures that in any location the radiation doses received by patients is consistent with that received in other locations in Australia and overseas.

An ionization chamber essentially isolates a certain volume of air and measures the ionization within that volume. However, the physical process of isolating a volume of air modifies the original photon and electron spectrum entering the volume. Thus, if the chamber is to act as a primary standard for calibration purposes, it is necessary to correct the measured ionization for the spectral perturbations due to the physical presence of the chamber.

The nature of the problem is such that not all correction factors can be determined experimentally. Within the field of radiation dosimetry, Monte-Carlo simulation programs, such as the EGS4 [16], have become an acceptable method of simulating the response of ionization chambers to photons and electrons and thus an alternative means of determining the appropriate correction factors. The absolute accuracy of the EGS4 package for simulating ion chambers is quoted at about 1% but relative accuracy is actually better than this.

One important correction factor for the primary standard, in the standardization of photons emitted by a cobalt-60 teletherapy source, is the extent of photon attenuation and scattering in the front wall of the chamber. Detailed measurements of chamber response as a function of front wall thickness have been performed and a quantitative model of interactions within the front wall used to determine the appropriate correction factors. These detailed measurements provide an interesting basis for comparison with Monte-Carlo, as not only the chamber but also the cobalt-60 photon spectrum are to be simulated. The calculations reported here concern the simulation of the chamber response as a function of front wall thickness. Accordingly, we need to run the model many times with different random number seeds and different front wall thicknesses.

Figure 3 shows the spreadsheet required to perform this calculation, including a chart showing the results. It is not possible to fully illustrate the complexity of this spreadsheet in the paper, however, it occupied 7 work sheets and contained many complex calculations and MS macros. Prior to the development of ActiveSheets, it was necessary for the results to be computed separately using EnFuzion, and for these to be transferred into the spreadsheet. With ActiveSheets, this could all be performed in the one spreadsheet, reducing the scope for error considerably.

Simulating Injecting Drug Users

Another project being conducted by the School of Business Systems at Monash University concerns the simulation of injecting drug users. Injecting drug use facilitates the transmission of numerous blood borne viruses such as The Human Immuno-deficiency Virus (HIV) and Hepatitis C Virus (HCV). Whilst these viruses are prevalent within the injecting drug using community, the process of transmission of the viruses from an infected individual to a susceptible individual is determined by various behavioural factors of the users as well as clinical factors of the viruses. From a public health perspective, intervention policies need to be directed at changing the injecting behaviour that facilitates this transmission. Given the inability to carry out controlled experiments in this field as well as the need to deal with a large number of parameters a simulation model makes a very effective analysis tool.

Considering the spread of HCV among injecting drug users (IDUs), some of the parameters in question are the prevalence of the virus, the infectivity of the virus, the frequency of injecting as well as the dynamics of the injecting process. A micro simulation model that enables researchers to create various cohorts of IDUs with differing injecting practices and prevalence levels and then experiment with various intervention policies is extremely useful. Given that spreadsheets are universally used across most disciplines, providing a spreadsheet interface to such a simulation program makes the model easier to use as well as more transparent to the user.

The results shown in Figure 4 were generated from a series of simulation designed to assess the impact of interventions that would result in changes to various parameters. They were gathered from the EnFuzion runs and then placed in a spreadsheet for charting and display.

Again, like the previous study, the work was originally performed using a conventional spreadsheet model and the runs were conducted using EnFuzion. However, ActiveSheets would have allowed us to combine these into one spreadsheet that computed and charted the results.

These two case studies illustrate that it is natural to consider the direct execution of a simulation from a spread sheet, since quite complex spread sheets have been created to specify the scenarios under consideration. The architecture that has been developed would allow these complex experiments to be created without complex external data manipulation, or for the need to use multiple tools.

RELATED WORK

There are a few commercial products for performing parallel Monte Carlo simulation using spreadsheets. However these are restricted to Monte Carlo simulation applications constructed using the supplied modeling functions. Palisade

has a product called @RISKAccelerator™; this supports the parallel evaluation of @RISK simulations [17]. Decisioneering has a similar product called Crystal Ball Turbo™ which parallelises their Monte Carlo simulation addin for Excel (Crystal Ball) [8]. However, neither system supports the general parallel evaluation of spreadsheet functions, as does ActiveSheets.

CONCLUSIONS

This paper has shown through some case studies that super computing using spreadsheets is highly desirable. To this end a mechanism for parallelising traditional spreadsheets has been defined. The mechanism is based on the idea of dataflow and is particularly useful in that it does not require modification to spreadsheet applications. The parallel spreadsheet tool interfaces with the popular parallel programming tools Nimrod and Enfuzion.

ACKNOWLEDGMENTS

This work has been sponsored by the Co-operative Research Centre for Distributed Systems Technology (DSTC), under the Australian Government CRC program. We wish to thanks Jon Giddy and Toby Chittenden for their contribution to the development of Active Sheets and Nimrod.

REFERENCES

[1] Abramson , D., Sosic , R., Giddy , J., Cope , M. “The Laboratory Bench: Distributed Computing for Parametised Simulations”, 1994 Parallel Computing and Transputers Conference, Wollongong, Nov 94, pp 17 - 27.

[2] Abramson D., Sosic R., Giddy J. and Hall B., “Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations”, The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[3] Abramson, D. “Parametric Modelling: Killer Apps for Linux Clusters”, The Linux Journal, #73, May 2000, pp 84 – 91.

[4] Abramson, D., Giddy, J. and Kotler, L. “High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?”, International Parallel and Distributed Processing Symposium (IPDPS), pp 520- 528, Cancun, Mexico, May 2000.

[5] Abramson, D., Power, K., and Sosic, R., “Simulating Computer Networks using Clusters of PCs”, HPC-TelePar'99 at the 1999 Advanced Simulation Technologies Conference (ASTC'99), April 11-15, San Diego, California,USA.

[6] Arvind, Bic, L. and Ungerer, T. "Evolution of Data-flow Computers", Chapter 1, "Advanced Topics in Dataflow Computing", Prentice Hall, 1991.

[7] B.J. Overeinder; L.O. Hertzberger and P.M.A. Sloot, “Parallel Discrete Event Simulation”, In W.J. Withagen, editor, The Third Workshop Computersystems, Faculty of Electrical Engineering, Eindhoven University, pp. 19-30. Eindhoven, The Netherlands, May 1991

[8] Decisioneering Inc. http://www.decisioneering.com/

[9] Dennis, J. "The Evolution of "Static" Data-flow Architectures", Chapter 2, "Advanced Topics in Dataflow Computing", Prentice Hall, 1991.

[10] Ferscha Alois, Tripathi Satish, “Parallel and Distributed Simulation of Discrete Event Systems”, TCS-TR-3336, University of Maryland, August 1994.

[11] H. Rajaei and R. Ayani, Design Issues in Parallel Simulation Languages, IEEE Design & Test of Computers, Dec. 1993, pp. 52-63.

[12] H.T. Mouftah, Distributed Discrete Event Simulation for Communication Networks, IEEE Journal on Selected Areas in Communication, Vol. 8, No. 9, pp.1723-1734, 1990.

[13] K. Motegi and S. Watanabe. A Parallel Scheduling of Multi-Step Diakoptics for Three Dimensional Finite Difference Method. IEICE Trans. Fundamentals, Vol.E76-A, No.10, pages 1822 - 1829, October 1993.

[14] Microsoft Corporation, Excel User’s Guide.

[15] Microsoft Corporation, Visual Basic for Applications.

[16] Nelson, W.R., Hirayama H. and Rogers, D.W.O. “The EGS4 Code System”, Stanford Linear Accelerator Center Report SLAC-265 (1985)

[17] Palisade Corporation, http://www.palisade.com/

[18] Paul Wonnacott and David Bruce, “Comparing the performance of a prototype implementation of the APOSTLE discrete event simulation language for shared and distributed memory multiprocessors”, pp. 30-36 in Proc. Object-Oriented Simulation 1998, San Diego, California, 11-14 January 1998.

[19] Richard Fujimoto, Parallel Discrete Event Simulation, Communications of the ACM, October 1990.

[20] Turbo Linux Inc, EnFuzion, http://services.turbolinux.com/shop/prod.php?id=enf

[21] Yi-Bing Lin, Bellcore Paul A. Fishwick, , “Asynchronous Parallel Discrete Event Simulation” CS-TR-95-005, University of Florida, 1999

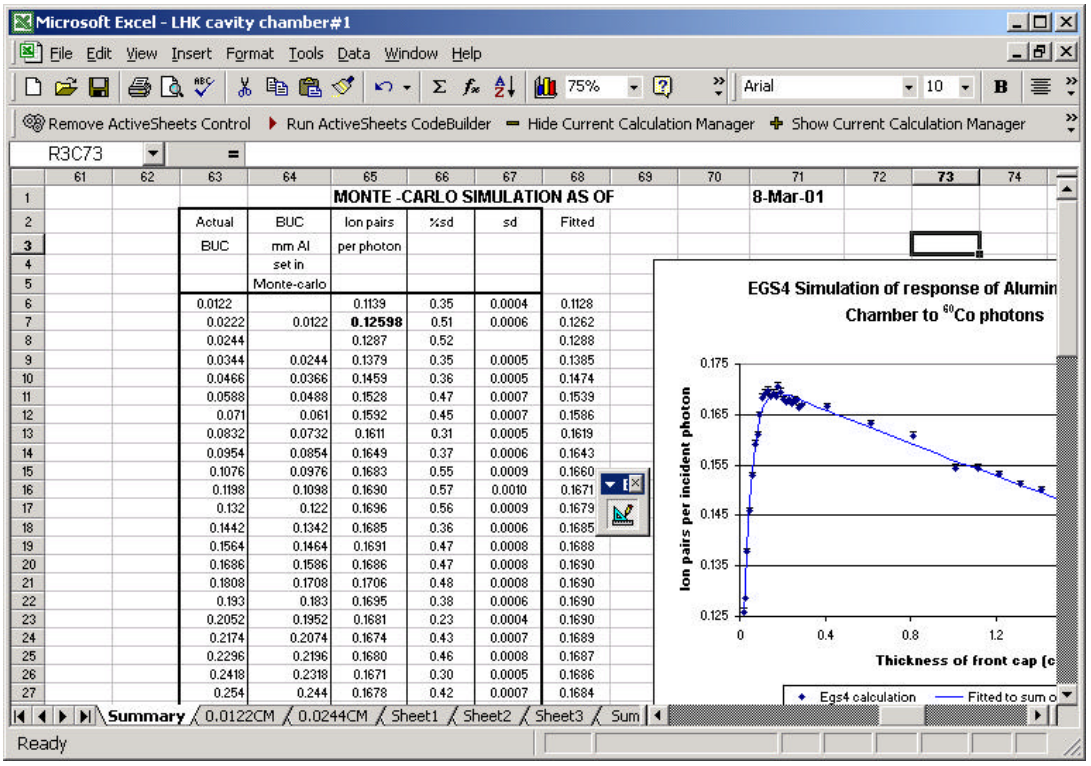


Figure 3 – Spreadsheet for ionisation chamber experiments

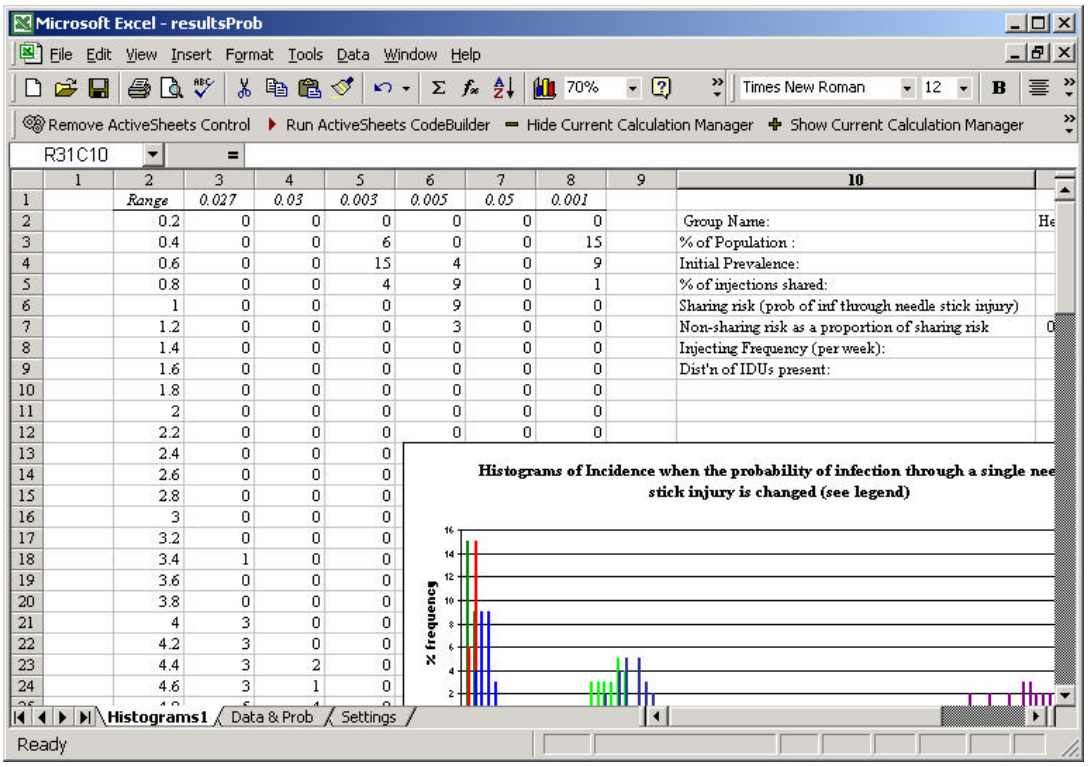


Figure 4- Spreadsheet for drug injecting simulation

Simplified Grid Computing through Spreadsheets and NetSolve

David Abramson[1], Jack Dongarra[2], Eric Meek[2], Paul Roe[3], Zhiao Shi[2]
[1]*Monash University, Australia*
[2]*University of Tennessee, Knoxville, USA*
[3]*Queensland University of Technology, Australia*
David.Abramson@infotech.monash.edu.au, dongarra@cs.utk.edu, meek@cs.utk.edu, p.roe@qut.edu.au, shi@cs.utk.edu

Abstract

Grid computing has great potential but to enter the mainstream it must be simplified. Tools and libraries must make it easier to solve problems by being simpler and at the same time more sophisticated. In this paper we describe how Grid computing can be achieved through spreadsheets. No parallel programming or complex tools need to be used. So long as dependencies allow it, formulae in a spreadsheet can be evaluated concurrently on the Grid. Thus Grid computing becomes accessible to all those who can use a spreadsheet. The story is completed with a sophisticated backend system, NetSolve, which can solve complex linear algebra systems with minimal intervention from the user. In this paper we present the architecture of the system for performing such simple yet sophisticated grid computing and a case study which performs a large singular value decomposition.

1. Introduction

Many have written about the potential of Grid computing. Yet it is only accessible to the few with expert knowledge on how to use this powerful resource. Real users want to use the Grid through simple and familiar tools such as Matlab™, Excel™, and web browsers.

In this paper we show how an end user tool, spreadsheets, can be used to drive the Grid through sophisticated libraries, in particular NetSolve [2]. We are able to use the Excel spreadsheet application to transparently drive the Grid. No special use need be made of Excel – nor is a special spreadsheet application required. Functions from NetSolve which are available on the Grid may be imported into Excel and used in spreadsheet formulae. Formulae are subject to data parallel evaluation and task parallel evaluation. That is, independent cells and formulae will be evaluated concurrently with no special effort on behalf of the user. Thus spreadsheets may be used to script the Grid. Spreadsheets are popular because they are easy to use and modify, and they support numerical data analysis without programming. Thus the Grid becomes accessible to all those who can use a spreadsheet.

A feature of modern spreadsheets is their capability to be extended; in particular custom functions may be incorporated into them, for example complex simulation functions and computational experiments. For these reasons spreadsheets make an ideal front-end for numeric computing since they support pre and post processing of data without the need for programming.

Existing spreadsheets, such as Microsoft Excel, have inbuilt sequential calculation mechanisms. Thus, they are unable to evaluate multiple cells concurrently using the standard execution mechanism in the calculation engine. To solve these problems we have constructed an architecture for Grid computing using Active Sheets [1,6] and NetSolve.

Active Sheets comprises a plug-in for Excel and a database (job manager) to cache results. It supports the asynchronous evaluation of spreadsheet formulae. This enables jobs to be generated from Excel and results to be received, and cached in the database for subsequent use. Results can be returned in any order and "downstream" calculations only fire when all of the inputs are available. Thus, the spreadsheet can be used to perform quite complex parallel computations. The database also has the side effect of allowing off-line evaluation.

NetSolve users normally have to write code, either in a conventional language like FORTRAN or C, or a

high level mathematic language like MatLab, to perform computations. Thus, the scripting of NetSolve functions needs to be specified programmatically. Using Active Sheets, meta-data which describes NetSolve functions can be downloaded and used to automatically generate proxy code for Excel functions. This enables web services to connect Active Sheets to NetSolve with no programming required. At the backend, the NetSolve system is able to schedule computation across NetSolve servers thereby minimizing any intervention needed by the user to drive the Grid.

This paper describes how a conventional spreadsheet may be evaluated in parallel without modifying Excel's core execution engine. The parallel evaluation mechanism is encoded using custom functions, and works in conjunction with the standard built in sequential evaluation mechanism of spreadsheets. The paper begins with an overview of Active Sheets and NetSolve. We then discuss an architecture for Grid computing using these systems. This is followed by a case study that illustrates the power of such a system.

2. NetSolve

NetSolve is a client-server system that enables users to solve complex scientific problems remotely. The system allows users to access both hardware and software computational resources distributed across a network. Some of the goals of the NetSolve project include ease-of-use for the user, efficient use of the resources, and the ability to integrate any arbitrary software component as a resource into the NetSolve system. Figure 1 shows the infrastructure of NetSolve and its relation to the application that use it. The shaded parts of the figure represent NetSolve system. We can observe that NetSolve acts as glue layer that brings application or user together with the hardware and/or software it requires to complete its useful work, which is the reason why systems like NetSolve are sometimes called grid middleware.

The major components of the NetSolve system are the *NetSolve agent*, an information service and resource scheduler, *NetSolve server*, a networked resource that serves up computational hardware and software resources, and the *NetSolve client* libraries, which allows users to instrument their applications with NetSolve calls for remote computational services. NetSolve agent is the gateway to the system. As an information service, the agent maintains a database of NetSolve servers along with their capabilities (including hardware performance and available software) and dynamic usage statistics. The agent attempts to find the server that will service the request the quickest using the information it possesses. It also balances the load amongst its servers and keeps track of failed servers.

The NetSolve server is the computational backbone of the system. A computer host becomes a NetSolve server when it is configured to run the NetSolve serve daemon. The server can run on single workstation, clusters of workstations, symmetric multi-processors or machines with massively parallel processors. NetSolve problem description file (PDF) is used to describe a computational problem formally. The PDF specifies basic information about a problem such as problem name, inputs and outputs. It also contains the information of libraries that implements any underlying functions or services being interfaced by NetSolve. In essence, the PDF defines a wrapper that NetSolve uses to call the function being incorporated

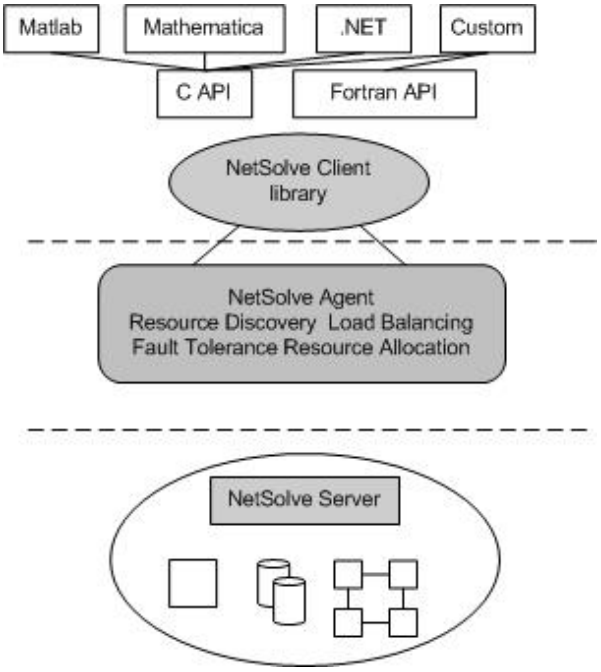


Figure 1. An architectural overview of NetSolve system

At the top tier of Figure 1, the NetSolve client library is linked with the user application. NetSolve supports program development in a variety of programming environments. Interactive environments such as Matlab [7] and Mathematica

[8] are easy to use and often relieve the user from details of variable declaration, memory allocation and other routine tasks. The programming interfaces for which NetSolve APIs have been developed are FORTRAN and C. Currently a native .NET interface is being developed, which will allow all native .NET applications to leverage the power of grid computing using NetSolve.

NetSolve continues to emerge as one of the leading programming paradigms for the Grid. Its light-weight and ease of use make it an ideal candidate for middleware and as it continues to evolve, the NetSolve system will be extended to become applicable to and even wider range of applications.

3. Active Sheets

Active Sheets features a component based spread sheet interface for specifying computational experiments. Figure 1 gives an overview of the Active Sheets vision. This allows computational experiments to be simply and easily driven from a spreadsheet.

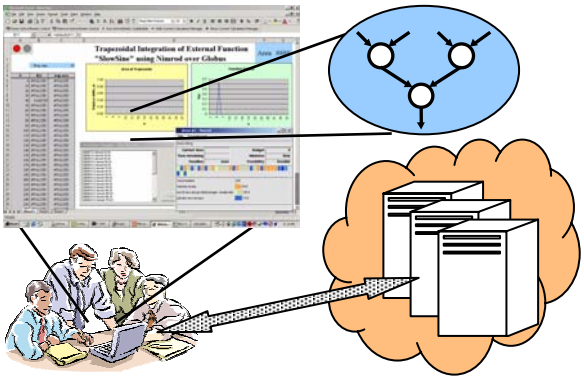
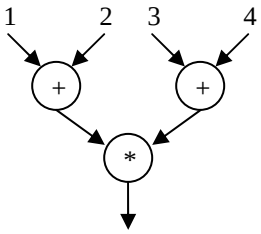


Figure 2: Active Sheets Overview

Backend computations are expressed through spreadsheet formula. These are automatically scheduled according to functional dependencies in a dataflow style, see Figure 3:. In addition operations themselves may be data parallel, implemented through Excel array formulae.



	A	B	C	D
1	1	2	3	4
2	=A1+B1		=C1+D1	
3	=A2*C2			

Figure 3. Data flow and spreadsheet equivalent

We have chosen to use Microsoft Excel as the base spreadsheet application, and have used Microsoft .NET plug-in to control Excel. The plug-in uses .NET remoting to connect to a local job manager, which caches results and manages connections to back-end machines such as NetSolve. Figure 4 shows the Active Sheets architecture.

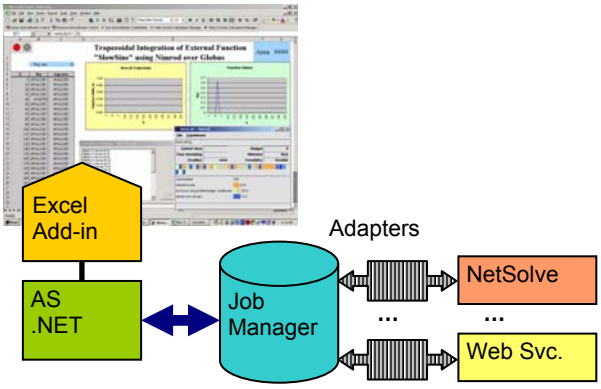


Figure 4. Active Sheets Architecture

The job manager caches results and hence decouples Excel from backend servers. This allows Excel to be used in a disconnected mode, and for the backend computations to be undertaken without Excel running. The job manager also supports a plug-in architecture for adapters; this enables different backend machines to be supported through corresponding adapters. The adapters are .NET assemblies which are dynamically loaded by the job manager.

4. Driving NetSolve from Active Sheets

We have constructed two options for connecting Active Sheets to NetSolve. The first is to produce a custom NetSolve adapter for Active Sheets which understands the native NetSolve communications protocol. This protocol operates over various TCP ports, and is in widespread use. The disadvantage of this is that it requires the designated TCP ports to be open and so does not always work across the Internet. The second approach is to create a web service wrapper for NetSolve using Microsoft .NET. This allows Active Sheets to drive NetSolve from any network so long as HTTP (port 80) is available. This is shown in Figure 5.

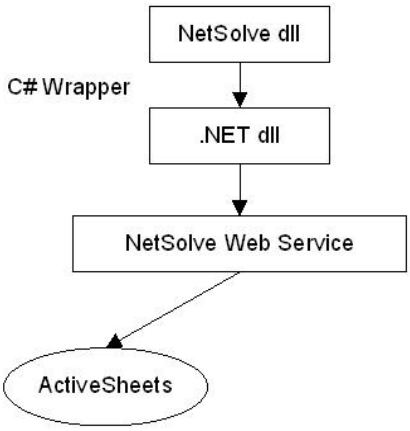


Figure 5. Active Sheets NetSolve Architecture

The system can be monitored at all points. Active Sheets has an option to show the jobs which are outstanding in the spreadsheet. The Job manager has a user interface which allows the status of all jobs, cached values, backend connections, and loaded adapters to be viewed. Finally a GUI tool enables the status of the NetSolve back-ends to be viewed.

5. An application: SVD

A SVD application has been developed to demonstrate the unique combination of NetSolve and Active Sheets. Singular Value Decomposition (SVD) [9] is a useful mathematical tool for finding and removing information stored in matrix form based on its significance to the rest of the data. SVD is widely used in the area of, digital signal processing [10], information retrieval [11] and analysis of gene expression [12].

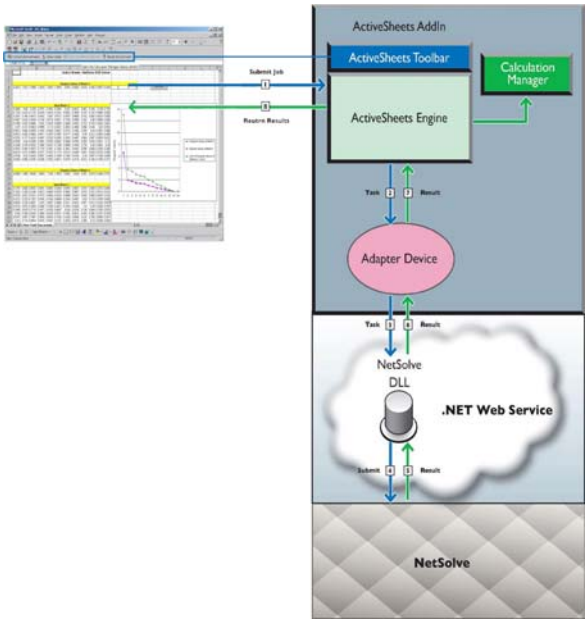


Figure 6. Usage Scenario of NetSolve-Active Sheets

Figure 6 shows the execution scenario of an SVD application using the Active Sheets/NetSolve system. Two matrices are laid out as separate tables in an Excel spreadsheet. When the user then submits a problem, Active Sheets dispatches and manages the tasks from the Excel spreadsheet to the back-end grid computing server(s). It also handles the completion of the tasks by ensuring the data is returned to the correct cell(s) in the spreadsheet. A plug-in adapter is used by Active Sheets to encapsulate the connection to the computational grid. When the user submits a problem and selects NetSolve as the computational grid, the Active Sheets engine will use the NetSolve adaptor to handle communication with NetSolve. The NetSolve adaptor will then forward all the user requests from the spreadsheet to the NetSolve servers in parallel. After the problems have been solved simultaneously and the results are ready, the NetSolve adaptor will retrieve the results from the NetSolve server(s).

The NetSolve adaptor contacts NetSolve through .NET web services [13] that have been implemented via SOAP over HTTP. The NetSolve web services are created for the sake of better interoperability and avoidance of firewall issues. Currently, the prototype

NetSolve web service implements *dgesvd* (a singular value decomposition routine from the LAPACK [14] mathematical library) in a non-blocking fashion. Invoking the *dgesvd* web method creates a new thread and immediately returns. The web service employs a .NET DLL which is built from an unmanaged NetSolve DLL. The unmanaged DLL takes care of all the underlining communication with NetSolve. The problem is then submitted to NetSolve using the *dgesvd* problem and solved. The result is returned to Active Sheets through the adapter and put into the correct cells. Finally, the singular values returned from the NetSolve server can be merged and placed in a results table in Excel.

If C was used to perform the same operation, a high level of programming proficiency would be required. For example, several hundred lines of nontrivial C code would be necessary to import the data, submit the data to NetSolve and finally merge the results. This is very complicated compared to the simple importing of data into, and the problem submission from, an Excel spreadsheet.

6. Related work

There are a few commercial products for performing parallel Monte Carlo simulation using spreadsheets. However these are restricted to Monte Carlo simulation applications constructed using the supplied modeling functions. Palisade has a product called @RISKAccelerator™; this supports the parallel evaluation of @RISK simulations [4]. Decisioneering has a similar product called Crystal Ball Turbo™ which parallelises their Monte Carlo simulation addin for Excel (Crystal Ball) [3]. Platform computing also have an Excel add-in for performing the data parallel evaluation of cells in its Platform Symphony™ software [5]. However, none of these systems supports the general parallel evaluation of spreadsheet functions, as does Active Sheets, nor do they support pluggable backend Grid computers.

7. Conclusions

This paper has shown how Grid computing may be simplified through an architecture which enables spreadsheets to use sophisticated numeric libraries through a web service enabled NetSolve. A mechanism for parallelizing traditional spreadsheets has been defined. The mechanism is based on the idea of dataflow and is particularly useful in that it does not require modification to spreadsheet applications. The parallel spreadsheet tool interfaces with NetSolve through both custom TCP protocols and web services. The latter uses a web service wrapper to web enable NetSolve. On going work concerns how to use large datasets resident outside Excel. For further information on Active Sheets and NetSolve see [1,2], both systems are available for download and use.

8. Acknowledgements

This research was funded in part by Microsoft research and by the Applied Mathematical Sciences Research Program of the Office of Mathematical, Information, and Computational Sciences, U.S. Department of Energy under contract DE-AC05-00OR22725 with UT-Battelle, LLC. We wish to thank Gavin Cheuk for his contribution towards the development of Active Sheets, and the DSTC (www.dstc.edu.au) for their continued support.

9. References

[1] Abramson, D., Roe, P., Kotler L and Mather, D., "*Active Sheets: Super-Computing with Spreadsheets*". 2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference, April 22-26, 2001, pp 110 – 115, Seattle, Washington (USA). http://www.citi.qut.edu.au/ActiveSheets/

[2] *NetSolve: Past, Present, and Future - a Look at a Grid Enabled Server*, S. Agrawal, J. Dongarra, K. Seymour, and S. Vadhiyar, Grid Computing: Making the Global Infrastructure a Reality, F. Berman, G. Fox, and T. Hey, eds., John Wiley & Sons, April 2003, ISBN 0-470-85319-0. http://icl.cs.utk.edu/netsolve

[3] Decisioneering Inc. http://www.decisioneering.com/

[4] Palisade Corporation, http://www.palisade.com/

[5] Platform Computing, "*Adapter for Excel*" http://www.platform.com/products/symphony/adapter/index.asp

[6] Abramson D., Sosic R., Giddy J. and Hall B., "*Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations*", The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[7] The MathWorks Inc. *Using MATLAB Version 5*. The MathWorks Inc., 1992.

[8] Wolfram S. The Mathematica Book(3rd edition). Wolfram Media Inc. and Cambridge University Press, 1996.

[9] Golub G., Van Loan C. F., *Matrix Computations*. The John Hopkins University Press. ISBN 0-8018-5414-8. 1996.

[10] Deprette, E. F, ed. *SVD and signal processing, Algorithms, Applications and Architectures*. North Holland. 1988.

[11] Furnas G.W., Deerwester S., Dumais S. T., Landauer T. K., Harshman R. A., Streeter L. A., and Lochbaum K. E. *Information retrieval using a singular value decomposition model of latent semantic structure.* In Proceedings of the Eleventh International Conference on Research & Development in Information Retrieval, pages 465-480, 1988.

[12] Wall, M.E., Dyck P.A., Brettin T.S. SVDMAN – singular value decomposition analysis of microarray data. Bioinformatics 6:566-568. 2001.

[13] Microsoft Web Services Developer Center: http://msdn.microsoft.com/webservices/

[14] *LAPACK Users' Guide, 3rd Edition*, E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammaring, A. McKenney, and D. Sorensen, SIAM Publication, Philadelphia, 1999, ISBN 0-89871-447-8.

PROVISIONAL SPECIFICATION FOR A PATENT

in the name of

DSTC PTY LTD

entitled

"PARALLEL EXECUTION MECHANISM FOR SPREADSHEETS"

Filed by:

FISHER ADAMS KELLY
Patent and Trade Mark Attorneys
Level 13
AMP Place
10 Eagle Street
BRISBANE QLD 400
AUSTRALIA

P/00/009
Regulation 3.2

AUSTRALIA

Patents Act 1990

PROVISIONAL SPECIFICATION

Invention Title: "PARALLEL EXECUTION MECHANISM FOR SPREADSHEETS"

The invention is described in the following statement:

PARALLEL EXECUTION MECHANISM FOR SPREADSHEETS

The invention relates to a parallel execution mechanism for spreadsheets. In particular, it relates to a method that implements parallel execution without modification to the spreadsheet sequential execution engine.

BACKGROUND TO THE INVENTION

Spreadsheets are popular because they are easy to use and modify, and they support numerical data analysis without programming. A feature of modern spreadsheets is the capability to be extended with custom functions or add-ins. For example, spreadsheets are particularly useful for complex simulation functions and computational experiments. For these reasons, spreadsheets make an ideal vehicle for numerical simulations since they support pre and post processing of simulation data without the need for programming.

Because simulations are computationally intensive it is desirable to execute as many of them in parallel as possible. This has previously been recognized and it is known to use parallel evaluation of complex numerical simulations using custom designed evaluation software. Once such software package is known to the inventors by the trade name Nimrod and is described in Abramson D., Sosic R., Giddy J. and Hall B., "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

Using a spreadsheet offers advantages compared to custom products such as Nimrod due to the generic nature of spreadsheets and their near ubiquitous availability and acceptance. A good example is the Microsoft Excel® product.

Products such as Microsoft Excel® have a sequential processing engine and cannot directly implement a parallel

processing mechanism without modification to the underlying processing engine. Modification of the underlying engine would result in a new product that would be very difficult to produce in a backwards compatible way. Thus modification of the processing engine is not a viable method of achieving parallel execution of calculations in known spreadsheet programs.

The desirability of enhanced calculations of formulas in spreadsheets has been recognized by Lotus Development Corporation. In United States patent number 5862400, a formula coprocessor is described for known personal computers. The benefit of the coprocessor with spreadsheet applications is that the result of one or more formulas can be cached for repeated use within a spreadsheet calculation. This invention speeds up spreadsheet calculation by eliminating multiple sequential evaluation of common formulas in a spreadsheet. The patent is concerned with a hardware solution to spreadsheet performance rather than in providing a mechanism for achieving parallel execution of spreadsheet calculations. The calculations in the Lotus patent are still sequential but known solutions are used instead of recalculation.

OBJECT OF THE INVENTION

It is an object of the invention to provide a method for parallel execution of at least some calculations in a spreadsheet.

Further objects will be evident from the following description.

DISCLOSURE OF THE INVENTION

In one form, although it need not be the only or indeed the broadest form, the invention resides in a method of parallel execution of spreadsheet calculations including the steps of:

defining custom functions that pass arguments and a function identifier to an evaluation process from a spreadsheet cell for parallel

evaluation of said custom functions;

constructing an evaluation table for storing interim and final results of said custom functions;

returning interim results to said spreadsheet cells during a first evaluation cycle;

forcing reevaluation of said spreadsheet cells; and

returning final value from said evaluation table to said spreadsheet cells.

Further features of the invention will be evident from the following description.

BRIEF DETAILS OF THE DRAWINGS

To assist in understanding the invention, preferred embodiments will now be described with reference to the following figures in which:

FIG 1 shows schematically the concept of parallel execution of a calculation;

FIG 2 shows how the calculation of FIG 2 is represented in a spreadsheet;

FIG 3 shows how the calculation of FIG 1 can be performed using parallel execution with custom functions;

FIG 4 shows an initial value of an evaluation table in the calculation;

FIG 5 shows the spreadsheet after a first calculation;

FIG 6 shows values in an evaluation table after a first calculation;

FIG 7 shows the spreadsheet after a first round of calculations;

FIG 8 shows the evaluation table at completion of calculations; and

FIG9 shows the spreadsheet at the completion of the calculation.

DETAILED DESCRIPTION OF THE DRAWINGS

In the drawings, like reference numerals refer to like parts. In FIG 1 there is shown a parallel depiction of the calculation (1+2)*(3+4). The result of the calculation is obviously 21. In the figure the nodes 1 represent operators. Once an operator has the required number of arguments it may be evaluated to generate a result which is output, possible to a next operator. The evaluation may be viewed as data flowing along arcs between operands.

Evaluation is sequential if the first operation is evaluated and the result stored, followed by the second operation being evaluated with the result stored and finally the third operation being evaluated using the result from the two earlier calculations.

In contrast the calculation can be performed far more efficiently in a parallel fashion if the initial two operations are evaluated concurrently followed by evaluation of the third operation.

The depiction of the calculation in a standard spreadsheet, such as Microsoft Excel® is shown in FIG 2. In the standard spreadsheet each cell is evaluated sequentially. Thus if a number is changed in the first row of the spreadsheet, each cell is reevaluated to obtain the changed result. In practice, the spreadsheet engine only evaluates cells which have changed since the last evaluation and any cell that depends on the changed cell. In a large spreadsheet this can be a very slow process.

Parallel evaluation of the formula in the spreadsheet can be achieved by defining custom functions, or add-ins, that work with the built-in spreadsheet functions. For the purpose of explaining the invention, two simple functions will be defined. The first function adfn(a,b) adds the arguments a and b. The second function prfn(a,b) gives the product of the arguments a and b. It should be appreciated that these functions are only for the purpose of example, in practice custom functions would not be defined for such simple functions, and

might well involve the execution of a program external to the spreadsheet program itself.

In order to evaluate an otherwise sequential spreadsheet in parallel a two stage evaluation process is adopted. In the first step the custom function sends its arguments together with a representation of the function to an evaluation process for parallel evaluation. An evaluation table is used to store the current state of the cell. The cell will be either unevaluated, under evaluation or evaluated.

The evaluation process may distribute the calculation to any number of processors, either within the same machine or externally, for evaluation. Mean time, the custom function returns an error or undefined value to the spreadsheet cell which prevents the spreadsheet from interpreting the cell as holding a valid value. By returning an error value immediately, the spreadsheet will continue sequential evaluation without waiting for the result of the custom function. This means that other custom functions in other cells can be evaluated by the same process. The spreadsheet will therefore complete a sequential evaluation cycle much more quickly than usual. In the mean time, the evaluation process evaluates the custom functions and stores the result in an evaluation table.

In the second step, the spreadsheet is forced to make a reevaluation. The error values in the spreadsheet are replaced by the values stored in the evaluation table by the evaluation process.

The cycles are repeated until all functions have been evaluated and there are no more changes in the spreadsheet.

The simple calculation shown in FIG 1 will be used as the basis of an example of the manner in which the invention may be put into effect. FIG 3 shows the spreadsheet of FIG 2 configured for parallel evaluation. The built-in functions used in FIG 2 have been replaced by custom functions in FIG 3. As mentioned above, custom

functions would not replace built-in functions. Custom function will be more complicated. Nonetheless, the simple example serves to explain the principle of the invention. In general, the custom functions may be arbitrary executable program which happen to take some parameters and return a result which can be stored in a spread sheet cell.

Persons skilled in the use of spreadsheets will realise that functions in cells A2, C2 and A3 will normally appear in a formula box with the function values appearing in the spreadsheet cells.

The spreadsheet starts be evaluating, for example, cell A2. The custom function adfn makes an "under evaluation" entry in the evaluation table to indicate that it is being evaluated. The functions arguments and its identifier are sent to the evaluation process for parallel evaluation. The function returns an undefined value to the spreadsheet. The spreadsheet has a value in the cell and therefore continues to sequentially evaluate cells. The custom function in cell C2 also makes an "under evaluation" entry in the evaluation table and returns an undefined value to the spreadsheet.

The custom function prfn requires the values from cells A2 and C2. It therefore makes an "unevaluated" entry in the evaluation table but returns an undefined value to the spreadsheet to allow the spreadsheet to continue with sequential evaluation.

The evaluation table at the end of the first evaluation cycle by the spreadsheet is shown in FIG 4. The spreadsheet appearance is shown in FIG 5.

Evaluation of the custom functions occurs in parallel in the background. After evaluation of the custom functions the evaluation table will contain the function values as shown in FIG 6.

The spreadsheet is then forced to perform a reevaluation. Upon reevaluation of the custom functions the values from the evaluation table will be returned so that the spreadsheet has the

appearance shown in FIG 7. Because the values in cells A2 and C2 have changed the spreadsheet will automatically reevaluate. By this time the parallel evaluation of the custom functions will have proceeded further so that the prfn function has been evaluated and the result stored in the evaluation table, as shown in FIG 8. The value for A3 will be returned to the spreadsheet when the spreadsheet seeks to evaluate the cell and the final result shown in FIG 9 will be obtained.

In many cases the parallel evaluation of the custom functions will be completed before the commencement of the second cycle of evaluation by the spreadsheet. In this case, the spreadsheet will not show the values shown in FIG 7. The evaluation table will have all of the values as shown in FIG 8 so the second evaluation cycle by the spreadsheet will result in the spreadsheet of FIG 9.

It will be appreciated that this simple example would require three evaluation cycles under sequential evaluation. Using the invention to achieve parallel evaluation, only two evaluation cycles are required. More importantly, the sequential evaluation does not wait for the result of a calculation before continuing the cycle. The inventors have found that this results in a considerable decrease in the time taken to evaluate a spreadsheet. The greater the degree of "parallelism" of the spreadsheet, the greater the improvement. Furthermore, it will be appreciated that if a spreadsheet has a high degree of parallelism, in other words a lot of cells that can be evaluated in parallel, multiple processors will provide an even greater improvement.

A number of techniques can be used to force the spreadsheet to perform a reevaluation. The technique used may be different for different spreadsheet products. There are three basic techniques. The first technique is to rely on the automatic evaluation process built-in to most spreadsheet programs. An automatic evaluation

occurs whenever a cell value changes. Thus simply changing the cell content each cycle will suffice because the spreadsheet engine will force a re-evaluation.

Another approach is to use command evaluation in spreadsheets where this facility is available. If a spreadsheet can be commanded to perform a complete reevaluation of all cells, the facility can be activated when the evaluation table shows all cells as having been evaluated.

Some spreadsheets can be commanded to reevaluate but only reevaluate cells that have changed since the last evaluation cycle. In this case the cell characteristics can be defined so that the cell reevaluates every cycle.

The invention operates with known sequential execution spreadsheets to provide improved performance through parallel evaluation for custom functions. Although a custom parallel evaluation spreadsheet can be designed, it is much more economic to provide an add-in for available spreadsheets.

Throughout the specification the aim has been to describe the preferred embodiments of the invention without limiting the invention to any one embodiment or specific collection of features.

DATED this twenty sixth day of June 2000

DSTC PTY LTD

By their Patent Attorneys

FISHER ADAMS KELLY

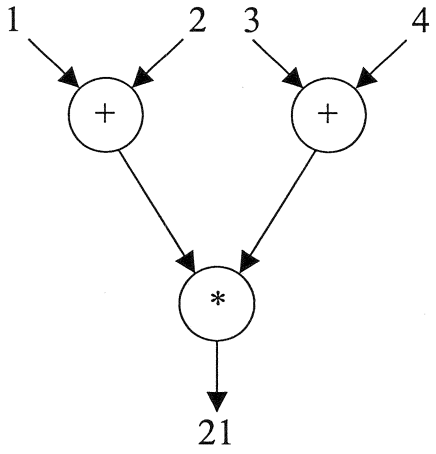


FIG 1

	A	B	C	D
1	1	2	3	4
2	=A1+B1		=C1+D1	
3	=A2+C2			

FIG 2

	A	B	C	D
1	1	2	3	4
2	=adfn(A1,B1)		=adfn(C1,D1)	
3	=prfn(A2,C2)			

FIG 3

Cell	State	Value
A2	Under Evaluation	Undefined
C2	Under Evaluation	Undefined
A3	Unevaluated	Undefined

FIG 4

	A	B	C	D
1	1	2	3	4
2	Undefined		Undefined	
3	Undefined			

FIG 5

Cell	State	Value
A2	Evaluated	3
C2	Evaluated	7
A3	Under Evaluation	Undefined

FIG 6

	A	B	C	D
1	1	2	3	4
2	3		7	
3	Undefined			

FIG 7

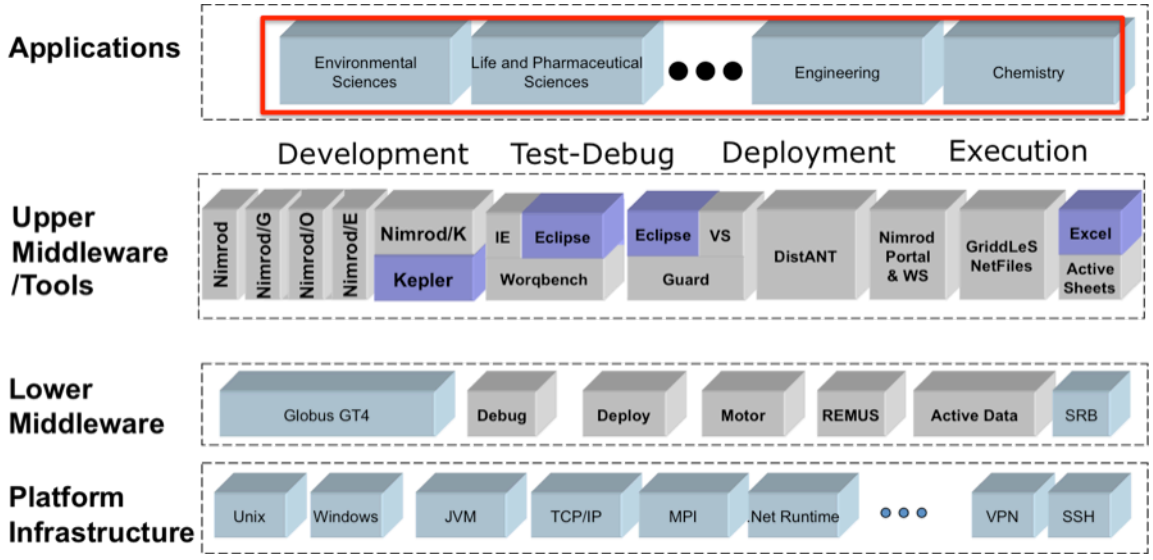
Cell	State	Value
A2	Evaluated	3
C2	Evaluated	7
A3	Evaluated	21

FIG 8

	A	B	C	D
1	1	2	3	4
2	3		7	
3	21			

FIG 9

Chapter 8: Applications



Many of the Nimrod papers contain application demonstrators, and these have already been discussed to date. In this chapter I highlight additional papers that single out particular applications.

(28) shows how parametric computing can support computer networking research. (29) and (32) demonstrate that the ideas can be applied to computational chemistry. (30) and (31) discuss the design of a virtual drug laboratory based on parametric computing. (33) and (34) discuss the solution of atmospheric models, and (35) solves an antenna design problem. (36) focuses on computational cardiology and (37) addresses structural biology.

(1) Abramson, D., Power, K., and Sosic, R., "Simulating Computer Networks using Clusters of PCs", HPC-TelePar'99 at the 1999 Advanced Simulation Technologies Conference (ASTC'99), April 11-15, 1999, San Diego, California, USA. 8-1 .. 8-9

(2) Baldridge, K., Sudholt, W., Greenberg, J. Amoreira, C., Potier, Y., Altintas, I., Birnbaum, A., Abramson, D., Enticott, C and Garic, S. "Cluster and Grid Infrastructure for Computational Chemistry and Biochemistry", "Parallel Computing for Bioinformatics", John Wiley & Sons, Inc, 2006. 8-10 .. 8-30

(3) Buyya, R., Branson, K., Giddy, J. Abramson, D., "The Virtual Laboratory: Enabling On Demand Drug Design with the Worldwide Grid", Concurrency and Computation: Practice and Experience, 15(1), 2003. 8-31 .. 8-49

(4) Abramson, D. A., Amoreira, C., Baldridge, K. K., Berstis, L., Kondrick, C., Peachey, T. C., 2006, A flexible grid framework for automatic protein-ligand docking, *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, 04 December 2006 to 06 December 2006, IEEE Computer Society, Los Alamitos USA, pp. 1-8. 8-50 .. 8-57

(5) Sudholt, W., Baldridge, K., Abramson, D., Enticott, C. and Garic, S., "Application of grid computing to parameter sweeps and optimizations in molecular modeling", Future Generation Computer Systems, 21 (2005), 27-35. Also appeared in International Conference on Computational Sciences, ICCS04, Krakow Poland, June 6 – 9, 2004. 8-58 .. 8-66

(6) Abramson, D., Kommineni, J., McGregor, J. and Katzfey, J. "An Atmospheric Sciences Workflow and its Implementation with Web Services", Future Generation Computer Systems, 21 (2005), pp 69 – 78. Also appeared in The International Conference on Computational Sciences, ICCS04, Krakow Poland, June 6 – 9, 2004. 8-67 .. 8-75

(7) Lynch, A. H., D. Abramson, K. J. Beringer, and P. Uotila (2007), Influence of savanna fire on Australian monsoon season precipitation and circulation as simulated using a distributed computing environment, Geophys. Res. Lett., 34, L20801, doi:10.1029/2007GL030879. 8-76 .. 8-80

(8) Lewis, A., Saario, S., Abramson, D. and Lu, J., "An Application of Optimisation for Passive 8-81 ..

RF Component Design", Conference on Electromagnetic Field Computation, Milwaukee, June 4-7th, 2000. 8-81

(9) Sher, A., Abramson, D, Enticott, C, Garic, S, Gavaghan, D., Noble, D., Noble, P., Peachey, T. "Incorporating local Ca2+ dynamics into single cell ventricular models using Nimrod/O", ICCS 2008, Krakow, Poland, June 2008. 8-82 .. 8-91

(10) Schmidberger, J., Bethwaite, B., Enticott, C., Bate, M., Androulakis, S., Faux, N., Reboul, C., Phan, J., Whisstock, J., Garic, S., Goscinski, W., Abramson, D., and Buckle, A., "High-throughput Protein Structure Determination Using Grid Computing", IEEE Workshop on High Performance Computational Biology (HiCOMB,www.hicomb.org), May 25, 2009, Rome, Italy, co-located with IPDPS 2009. 8-92 .. 8-99

Contributions

Most of the co-authors on these papers have contributed their expertise on particular applications.

Simulating Computer Networks using Clusters of PCs

David Abramson †
Kieran Power †

Rok Sosic ‡

†School of Computer Science and Software Engineering
Monash University
Clayton, Vic 3168.
davida@csse.monash.edu.au

‡Griffith University
Nathan, Qld 4111.

Abstract

Simulation has become the dominant technique for evaluating the performance of many products before real hardware is produced. In computer networks, this is particularly true, and can avoid expensive hardware and software development. Using simulation it is possible to explore different protocols and design options, under different loads, long before the design is frozen. However, such simulations can be time consuming, especially if a large number of scenarios are to be considered. In this paper, we show how it is possible to perform distributed execution of a network simulation written for the OpNet package. This is performed using a distributed computing tool called Clustor. Very high efficiency has been experienced on a variety of workstation clusters. A case study is presented in which a real network protocol is evaluated using OpNet and Clustor.

Keywords: Computer Network Simulation, Parallel Discrete Event Simulation, Clusters.

1 Introduction

Today's network products are becoming increasingly complex. The wide variety of networking protocols and transmissions methods, means that a single product may have to support many different protocols concurrently. Importantly, as new protocols are developed for different transmission mediums, it is important to simulate the behavior of these algorithms before they are implemented in real products. Simulation has emerged in this area as an extremely important technique for both verifying the correctness of a protocol, but also for studying how it behaves under different operational parameters, such as network delays and user demand.

Simulation can be expensive. In order to model a given real world scenario accurately, the simulation must be performed with sufficient detail to capture real hardware behaviour and user demand. This is usually achieved by a discrete event simulation at a low level, and it may take a long time to gain a realistic summary across different scenarios, particularly if monte-carlo methods are employed to study stochastic behaviour.

There have been many attempts to accelerate discrete event simulation by parallel processing. Because the basic process is sequential, there has only been limited success in accelerating the execution of a single scenario. However, when multiple independent scenarios are simulated, it is possible to execute them concurrently – this is regarded as an "embarrassingly parallel" application. In spite of this, there has not been wide acceptance of parallel and distributed platforms for simulation systems. This is most likely because the cost of adding and maintaining the parallel code in the already large simulation system is too high for the expected returns.

In this paper we describe a way of achieving the benefits of parallel computing for evaluating independent scenarios, without any modification to the underlying simulation engine. We make use of the Clustor [12] package, which can execute a program multiple times on many platforms, changing parameter values between executions. Because Clustor is generic, it can actually be applied to a wide range of applications, thus reducing the cost of parallel computing infrastructure for any one applications.

The paper contains a case study in which a new network protocol is considered, and the results of both the simulations, and the parallel computation efficiency, are presented.

2 Parametric Modelling on Clusters of PCs using Clustor

2.1 Affordable Supercomputing

Traditionally, supercomputers have been built with high performance processors, which use techniques to execute as many computations concurrently as possible. These machines have employed a number of arithmetic units, and sufficient hardware to allow them to be fed with data and instructions at their peak rate. Supercomputers built as parallel processors have also contained special shared

memory or message passing hardware as an integral part of the machine design – this special hardware, in combination with the high speed processors has meant that they have been expensive. However, the rapid change in the price performance characteristics of PCs and high speed networking hardware has made it possible to assemble a super-computing platform from large clusters of PCs. A number of projects around the world have been investigating this approach [3] [8][11], in which PCs are simply racked together and a high speed switch, such as a Myirinet [23][14] is provided for inter-processor communication. In this paper we explore one particular application of machine clusters, namely "parametric modelling".

2.2 Parametric Modelling on PC Clusters

Parametric modelling involves running the same application many times, varying one of more parameters each time the code is executed. Because this type of application is highly parallel, it is possible to run more than one copy of the code at the same time using a machine cluster. However, managing an experiment of many thousands of runs can be time consuming and error prone. Figure 1 shows a sample machine configuration for a machine cluster. The processors are configured without display monitors, and each one contains a reasonable amount of local disk. A "root" machine is responsible for controlling the cluster, and may contain the user file system. We do not assume that the cluster processors share a file system, although this is not excluded.

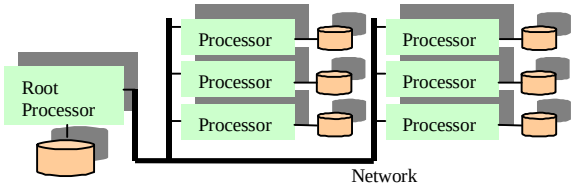


Figure 1 – Cluster configuration

Performing a computational experiment involves 5 distinct phases:

1. Experiment pre-processing
2. Execution pre-processing
3. Execution
4. Execution post-processing
5. Experiment post-processing

Phases 1 and 5 are performed once per experiment. In these, data is set up for the experiment and results are processed. In phase 5 it is often necessary to run data interpretation or visualisation software. Phases 2, 3 and 4 are run for each distinct parameter set. Phase 2 prepares data for a particular execution, and phase 4 reduces the data from an individual execution. Phase 3 actually executes the program given a set of parameter values. During phases 2 and 4 files may be moved between the root machine and the cluster processes – unlike general parallel computing this is the only communication that occurs between tasks.

Because of the repetitive nature of these computations, it is desirable that they are performed automatically. Nimrod [7][4][6][5] and Clustor [12] are general purpose tools which automate this process. Thus, it is possible to run thousands of jobs per experiment without consideration of the underlying computational resources, and without concern for management of the data.

2.3 Active Tools Clustor

Clustor is a tool which manages the execution of parametric studies across distributed computers. It takes responsibility for the overall management of an experiment, as well as the low level issues of distributing files to remote systems, performing the remote computation and gathering the results. Clustor is a commercial version of the research system Nimrod, and has been applied to a number of case studies.

Clustor supports the five phases of an experiment discussed in the previous section. A user develops a short "plan" file which describes the parameters, their default values, and the commands necessary for performing the work. The system then uses this information to transport the necessary files and schedule the work on the first available machine.

When the user invokes Clustor on a workstation, the machine becomes known as the "root" machine because it controls the experiment. When the dispatcher executes code on remote platforms, each of these is known as a computational "node". Thus, as shown in Figure 2, a given experiment can be conducted with one root and multiple nodes, each of a different architecture if required.

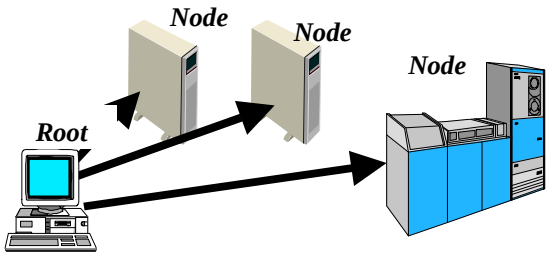


Figure 2 – Clustor Configuration

A plan file is composed of two main sections, the parameter section and the tasks section. Figure 3 shows a sample plan used for some of the simulation work discussed in this paper. The experiment consists of 4 parameters, "grpval", "conval", "seedval" and "mttfval". "grpval" and "mttfval" are selected from a list of possible legal values. "conval" is taken from a range of values, varying from 0.5 to 1.0. "seedval" is a random integer between 0 and 10,000 – in this case each execution receives a different seed value. Clustor generates one job for each unique combination of the parameter values, by taking the cross product of all values. In the plan in Figure 3, up to 162 jobs would be generated.

In this example there are two tasks – "main" and "rootfinish". The "main" task is executed for each set of parameters. It runs an OpNet simulation on a node, passing the parameter values to the program through the command line. It then copies a number of result files back to the root machine, appending each name with a unique identifier. This task corresponds to the 3rd phase discussed in the previous section. The "rootfinish" task executes once per experiment, and corresponds to the 5th phase described in the previous section. It catenates all of the output files together to produce a single list of results for each observed variable.

```
parameter grpval label "Group Value" float select
        anyof 0.7 0.5 0.2;
parameter conval label "Connectivity" float range
        from 0.5 to 1.0 step .1;
parameter seedval label "Seed Value" integer random
        from 0 to 10000;
parameter mttfval label "MTTF" integer select
        anyof 200 500 800 1100 1400 1700 2000 2300 2600 2900;
task main
        node:execute op_runsim -net_name adhoc -duration 30000 \
        -seed $seedval -ef attrs_1 optimize_simulation \
        -mttf $mttfval -connectivity $conval -grp_size $grpval \
        -join_rate 0.1111 -core_node 5-static/dynamic static
        copy node:average_delay average_delay.$jobname
        copy node:hop_count hop_count.$jobname
        copy node:overhead_result overhead_result.$jobname
        copy node:throughput_result throughput_result.$jobname
endtask
task rootfinish
        execute cat average_delay.* > average_delay
        execute cat hop_count.* > hop_count
        execute cat overhead_result.* > overhead_result
        execute cat throughput_result.* > throughput_result
endtask
```

Figure 3 – Sample Plan file

The plan file is processed by a tool called the "generator". The generator takes the parameter values, and gives the user the choice of actual values. It then builds a "run" file, which contains a description for each job. The "run" file is processed by another tool called the "dispatcher", which is responsible for managing the computation across the nodes. The dispatcher implements the file transfer commands, as well as the execution of the model on the remote node.

3 Simulation of Network Protocols using OpNet

The OpNet [13] simulation package allows rapid modelling of complex communication systems by providing features such as: a graphical user interface (GUI), a comprehensive library of contemporary protocols (e.g., TCP/IP, ATM, and Ethernet) and an object oriented development environment.

The process of modelling a communication system is distributed over a three level hierarchy. The lowest level is referred to as the *process model* and is the point at which a protocol's finite state machine (FSM) is defined. The middle level is called the *node model* and contains one or more process models. When multiple process models are present they are usually configured to represent a layered protocol stack. Finally, the *network model* is the highest level and contains the individual elements that make up a real network (e.g., routers, clients and servers). Each of these elements is associated with a single node model that determines its behavior in the network.

Currently OpNet provides two methods for executing simulation runs: firstly, from within the OpNet environment and secondly, as an independent executable. The GUI of the OpNet environment provides a convenient method for subjecting a simulation model to a range of input parameter values. The downside associated with the OpNet environment is that simulations can only be executed sequentially on a single machine. When the range of input parameter values is small the time required to complete all simulations maybe acceptable, however for a large range of input parameter values the time required to complete all simulations may become prohibitively long.

By using the independent execution technique, a reduction in simulation time can be obtained. To create an independent executable, OpNet compiles the simulation model into an executable program that is run like any other program on a computer. By running the program on different computers (using unique input parameter values) concurrent execution of the simulation can be carried out. Obviously, the more computers that are used the greater the reduction in simulation time. Unfortunately, using independent executions does have some drawbacks, probably the most significant of which is that it makes inefficient use of computer resources.

Typically, a number of simulation runs are distributed equally across available computers. However, as is often the case with computers, their performance can vary considerably due to differences in hardware (e.g., CPU and memory) or, in the case of shared systems, resource usage. This means that some computers will complete their quota of the simulations prior to others and will remain idle even though the slower computers may have numerous simulation runs pending. Clustor distributes the load across the machines using a FIFO scheduling algorithm, which accounts for the varying run times by using the next available processor when a new task is executed.

Another shortcoming associated with independent executions is that each simulation on a different computer normally needs to be managed manually. This requires a user to log into each of the computers being used to run simulations and provide it with a copy of the simulation executable, then initiate the simulation run. In a situation where only a few computers are being used this process may be manageable, but if many computers are being used the process of manually logging into each of the computers may become tedious. Clustor avoids these problems by automating the process completely.

4 Simulating the 'Ad hoc Multicast Network Protocol': A Case Study

4.1 Prior Work

With the demand for multiparty services such as audio/video conferencing and replicated databases, it is imperative that today's computer networks support multicast routing. Multicast routing provides an efficient technique for the dissemination of a single packet to multiple destinations. The past decade has seen an enormous amount of research into multicast routing, culminating in the development of numerous protocols: DVMRP [9], MOSPF [16], CBT [1], PIM [25] and MIP [21]. These protocols have successfully been deployed in the Internet and currently form the basis of the MBONE [2].

Unfortunately, the majority of today's multicast routing protocols have been designed to operate in networks such as the Internet where nodes (routers) are interconnected via relatively high bandwidth hardwired links (e.g., coaxial and fiber optical cable) and changes in the network topology occur infrequently. In contrast, an ad hoc network contains no fixed infrastructure and nodes are interconnected via relatively low bandwidth wireless links. In addition, nodes are free to roam in a random fashion resulting in a network topology that changes in a rapid and unpredictable manner.

In an ad hoc network, the time and communication complexity associated with protocols developed for the Internet may lead to the generation of excessive control overhead and/or the inability to quickly re-establish routes. These problems have prompted the recent development of two multicast routing protocols [24][22] specifically for use in an ad hoc network. However, these two proposals lack several desirable properties and in certain situations may not perform particularly well.

The AMP protocol has been developed to address shortcomings associated with current ad hoc multicast routing protocols. Its operation is distributed, provides both source and receiver initiated tree construction, allows dynamic group membership and is loop and deadlock free.

4.2 The AMP Protocol

While a complete description of AMP is beyond the scope of this paper, the following provides an overview of AMP's operation. The interested reader can consult [17][18] for further details.

To provide an efficient multicast routing service AMP uses diffusing computations [10] to structure the underlying network as a directed acyclic graph (DAG) [20], upon which a multicast distribution tree is constructed. Additionally, protocol state maintenance is performed using a hard-state mechanism. The main benefits to be gained from these techniques is that topology update packets are only exchanged when required (as opposed to periodically), and in the event that topology update packets are exchanged, they are generally confined to a small region and need not be forwarded throughout the entire network.

Nodes running AMP maintain several parameters: height, DAG link array, multicast link array and a list of multicast groups active on the underlying DAG. Each of the links interconnecting a node with its neighbouring nodes are assigned a direction of either 'upstream' or 'downstream'.

The direction assigned to a link forming the DAG is obtained by comparing a node's height to the height of its neighbours. If a node's height is greater than its neighbour's then a link is assigned a direction of downstream. On the other hand, if the node's height is lower than its neighbour's then a link is assigned a direction of upstream. Once a link forming the DAG has

been assigned a direction it is then stored in the DAG link array. The direction of a multicast link is simply determined by the exchange of multicast control packets. When a node sends the appropriate multicast control packet to a neighbour, the corresponding multicast link is assigned a direction of downstream. The recipient of the multicast control message assigns the direction of upstream to the multicast link. The direction of a link forming part of the multicast tree is stored in the multicast link array.

To commence a multicast session a node first sets its height to 0 and then forwards a join request packet to each of its neighbours; the join request contains several parameters among which is the height of the node sending it. On receiving a join request, a node first determines if the packet has previously been received, if so the packet is simply discarded. If not the node updates its height to that contained in the join request packet plus one (height + 1). The direction of each of the DAG links is then updated as discussed previously.

Once a node has received a join request packet it is free to join the multicast session. To do so, a node first selects a downstream link from the DAG link array and then proceeds to forward over it a join response packet. The node sending the join response assigns a multicast direction of upstream to the link over which the join response was sent. On receiving a join response packet a node first assigns a multicast direction of downstream to the link over which the join response was received. The node then determines whether it is already a member of the group specified in the join response packet. If so, the packet is discarded. If not, the node simply joins the multicast session as previously discussed.

On receiving multicast data a node determines the direction of the multicast link over which the data was received. If the direction is downstream then the data is discarded, however if the direction is upstream then the data is forwarded to the application and/or interested end systems (hosts). In addition, if the node contains downstream multicast links the data is forwarded onto its neighbours.

When a change in the network topology occurs, more specifically when a link fails, a node reacts in several ways. If the failed link does not result in either: a node loosing its last downstream DAG link or loss of an upstream multicast link, then the node takes no action. However, if the failed link results in a node loosing its last downstream DAG link then the node sets its height to a level that is higher than the neighbour with the highest height (this technique is referred to as link reversal). The node then assigns the direction of downstream to all DAG links and informs its neighbours of its new height by forwarding an update packet. On receiving an update packet a node updates the DAG direction of the link over which the update packet was received. On receiving an update it is possible that a node will also loose its last downstream DAG link. When this does occur, a similar procedure to the link reversal is employed, however the DAG direction of the link over which the update was received is not altered.

When the link reversal process is carried out it is possible that a link will have a DAG direction and multicast direction that are the same. To ensure the multicast tree remains correct (loop free) a link's DAG and multicast direction must be opposite. During the link reversal process a node removes any entries from the multicast link array. On receiving the update packet a node follows a similar procedure and removes any multicast link entries that have an equivalent DAG direction. In doing this it is more than likely that a node will no longer remain part of the multicast group. To rectify this situation a node simply rejoins the multicast session.

4.3 A Simulation model of AMP

Using the OpNet simulation package, a comparative simulation study of AMP and RBM was conducted. The simulated network consists of 48 nodes (routers) interconnected via point-to-point links with a bandwidth of 1024 bits/sec, resulting in a mesh topology as shown in Figure 4. The size of packets exchanged between nodes is 1024 bits for data while control packets vary between 10 and 60 bits depending on the type of control packet. Data packets are generated at a constant bit rate, the value of which is a simulation input parameter.

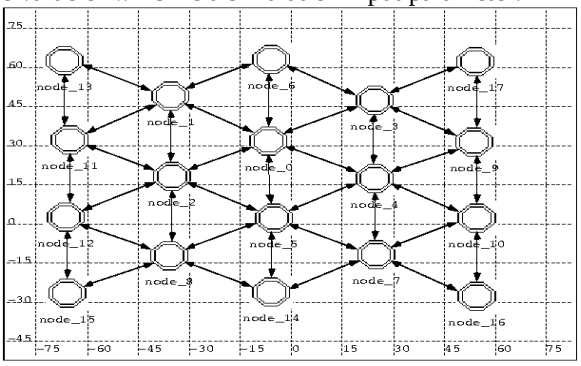


Figure 4 – Example Network.

To simulate the dynamic nature of an ad hoc network each point-to-point link alternates between two states

(active and inactive) according to a two-state Markov process. In the active state packets are transferred in order and error free. Any packets in the process of being delivered when an active link becomes inactive are destroyed. In the inactive state all packets are destroyed.

The duration a link spends in the active state is exponentially distributed with the mean being a simulation input parameter. A high mean value corresponds to a relatively static topology while a low mean corresponds to a highly dynamic topology. The time a link spends in the inactive state is determined using the following equation:

$$1/\lambda = ((1 - \alpha) / \alpha) * 1/\mu$$

where $1/\mu$ is the average duration a link remains in the active state and α (a simulation input parameter) is the average fraction of total time a link remains in the active state. A high α corresponds to a densely connected network, while a low α corresponds to a sparsely connected network.

At the being of a simulation run both the core node and multicast group members are randomly selected. To observe the scaling properties of both protocols the number of group members is varied, the value of which is a simulation input parameter. Each simulation is run for a duration of 30000 seconds. Table 1 summarizes the simulation input parameters.

Input Parameter	*Description*	*Value*
Link Active Time (Sec)	Represents the average time a link remains active.	200 to 4200 in steps of 500
Connectivity	The average fraction of links which are active at any given instant in time.	0.3, 0.5, 0.7 and 0.9
Group Size	The fraction of nodes which are members of the multicast group.	0.1, 0.4 and 0.7
Data Rate (Pkts/Min)	The rate at which data packets are generated.	6, 3, 1.5 and 1

Table 1 – Simulation Input Parameters.

5 AMP Simulation Results

The simulation model described in the previous section (4.3.A) was used to collect data in relation to the performance metrics summarized in Table 2.

The following results show the performance of AMP and RBM for a connectivity of 0.9 and a data rate of 6 Pkts/Min.

Metric	*Description*
Throughput (Bits/Sec)	Average number of data bits received by multicast group members per second.
Overhead (Bits/Sec)	Average number of control bits generated by all network nodes per second.
Hop Count	Average number of hops required to distribute data packets to multicast destinations.
Packet Delay (Sec)	Average time required to deliver data packets to multicast destinations.

Table 2 – Performance Metrics.

5.1 Throughput

By examining Figures 5 and 6 it can be seen that for a small group size (0.1) there is little difference between the throughput of AMP and RBM. However, for larger group sizes (0.4 and 0.7) AMP's throughput is significantly higher than that of RBM. More importantly, AMP's throughput remains relatively constant with respect to the rate of topological change (link active time). In contrast, RBM's throughput drops significantly as the rate of topological change increases (link active time < 1500).

5.2 Overhead

Figures 7 and 8 show that the overhead generated by AMP is considerably less than that generated by RBM. Interestingly, the overhead generated by AMP is largely independent of the group size whereas the overhead generated by RBM is closely related to the group size. This characteristic has important implications in determining the protocol's ability to scale.

5.3 Hop Count

As shown in Figures 9 and 10, AMP requires less network resources than that of RBM. For a large group size (0.7) the number of links forming a multicast tree constructed by AMP is roughly equal to the number of links used by RBM in a small group (0.1). As the group size increases, RBM uses significantly more links than AMP.

5.4 Packet Delay

The average delay encountered in delivering data packets to multicast destinations is shown in Figures 11 and 12. In a relatively static network (link active time > 2000) with a small group (0.1) AMP and RBM have similar delays. However, as the rate of topological change increases (link active time < 2000) AMP's delay becomes less than RBM's. For larger group sizes (0.4 and 0.7)

AMP's delay is significantly less than RBM's over the entire range of link active times.

6 Clustor Performance

We have experimented with two different configurations in the application of Clustor to the AMP simulation reported in the last section. Initially, the work was performed on a small number of SUN Ultra Sparc workstations, connected by 10 Mbit ethernet and a NFS mounted file system. The results presented in this paper were produced using this environment. We also experimented with running the simulations on a 20 processor Windows NT cluster consiting of 10 dual processor, 350 MHz, Pentium IIs. Unfortunately, at the time the paper was written the OpNet binaries were not running correctly on Windows NT, and so it was not possible to perform the same simulation runs. To gauge the effectiveness of the platform, however, we performed the same experiment without the actual network simulation code. All of the input files were copied to a node, and the output results were copied back to the root. However, instead of the OpNet simulation another null process was executed lasting 25 minutes – well short of the 8 hours for the full simulation on the Sparc Stations. This scheme simulates the effect of running the actual experiment, shows the effect of the file copying on the performance and gives a worst case performance. Table 3 shows the performance of the resulting simulations. In both cases the efficiency is very high, mainly because the simulation time far exceeded the setup and management times.

System	Time on 1 processor	Time on N processors	Speedup/ Efficiency
SUN Ultra Sparc	480 mins	3 procs 160 mins	3/ 100%
Pentium II Cluster	1750 mins	20 procs 100 mins	17.5/ 87.5%

Table 3 – Speedup using Clustor

7 Conclusions

This paper has described how it is possible to accelerate the parametric simulation of computer networks using a sequential commercial-off-the-shelf simulation package, OpNet, and a cluster of inexpensive PCs. The key is to use a software tool, called `Clustor`, for distributing the individual runs for each parameter combination concurrently to as many machines as possible. Clustor administers the generation of individual jobs and their distribution in a seamless way, so the user is unaware that the jobs are not running on the one machine. Using this technique it is possible to reduce the execution time dramatically.

The paper also presented some simulation results for a particular network protocol, AMP, which has been designed to support mobile communication multicast operations. The paper shows some results of the simulations that have been performed, and shows how the protocol performs as key parameters, such as rate of topological change, traffic load, connectivity and group size are altered.

A new research version of Clustor, called Nimrod/G is currently being prepared [5]. Nimrod/G addresses issues of how to access wide area resources using a meta-computing toolkit, called Globus [15]. Nimrod/G also presents an interface which allows the user to specify soft real time deadlines for the experiment. In this way the user can request that a given experiment will be completed by a particular time. The system then searches for sufficient resources to perform the work, and schedules the execution on the machines. This type of tool will make it possible for users to perform very large parametric modelling experiments using widely distributed computers. Experiences with such a tool will be reported at a later date.

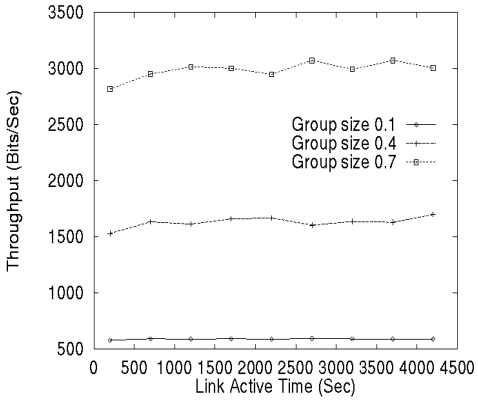


Figure 5 – AMP Throughput.

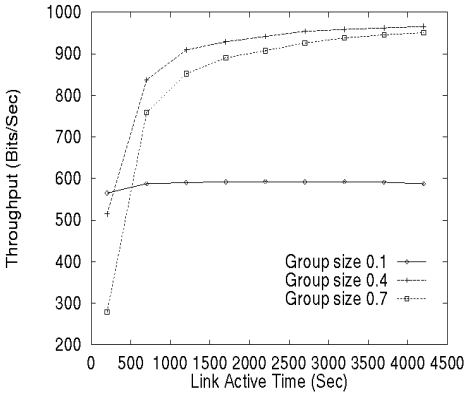


Figure 6 – RBM Throughput.

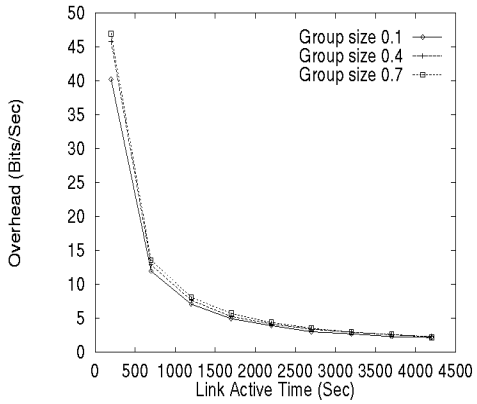


Figure 7 – AMP Overhead.

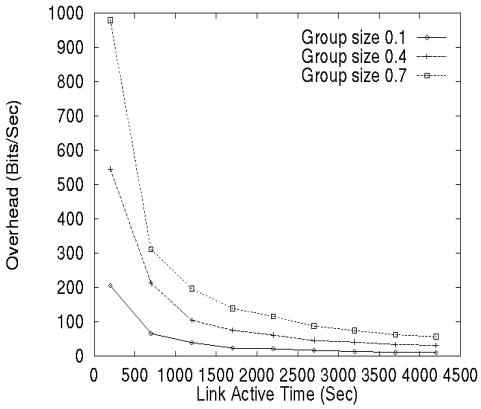


Figure 8 – RBM Overhead.

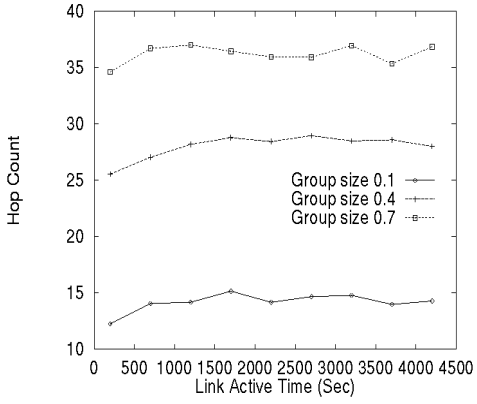


Figure 9 – AMP Hop Count

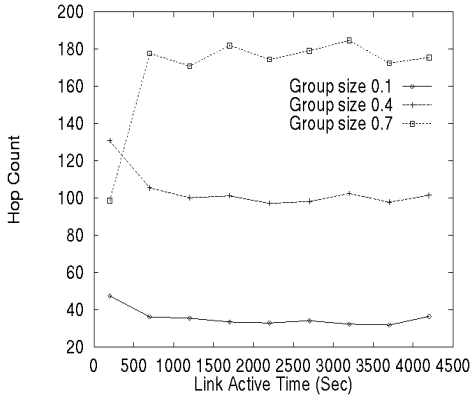


Figure 10 – RBM Hop Count.

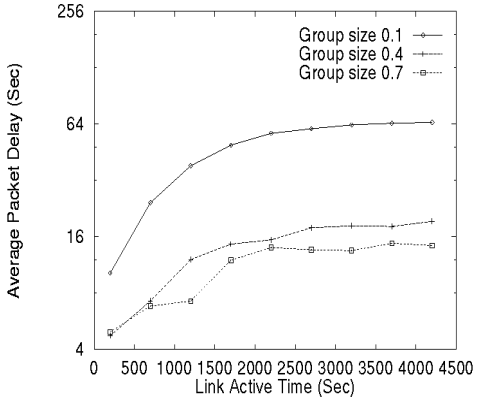


Figure 11 – AMP Packet Delay

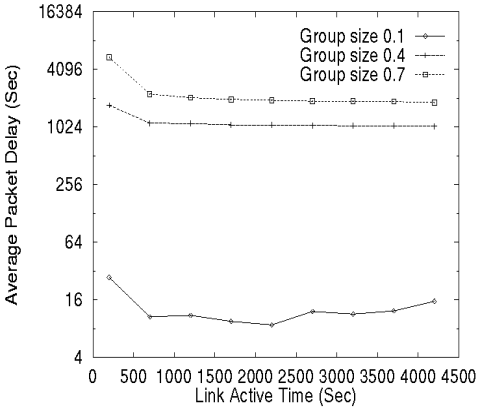


Figure 12 – RBM Packet Delay

8 Acknowledgements

The authors would like to thank Mr James Aarons and Mr Johnny Darous, who performed this work as part of their third year Computer Science project at Monash University, and Russell Kennett who was employed on a summer scholarship. We also acknowledge support from the Monash University Research Fund for providing the

computational resources, and Active Tools for providing the Clustor package. Nimrod was developed as part of a research program supported by the Co-operative Research Centre for Distributed Systems Technology. Bin Qiu, Mahbub Hasan and Jim Breen contributed to the design of the AMP protocol.

9 References

[1] A. J Ballardie, P. F. Francis and J. Crowcroft. Core-based trees (CBT). *Proc. SIGCOMM '93*, Pages 85-95.

[2] C. Huitema. *Routing in the Internet*. Prentice Hall PTR, 1995.

[3] D Ridge, D Becker, P Merkey, T Sterling, "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs", Proceedings, IEEE Aerospace, 1997.

[4] D. Abramson , R. Sosic , J. Giddy , M. Cope, "The Laboratory Bench: Distributed Computing for Parametised Simulations", 1994 Parallel Computing and Transputers Conference, Wollongong, Nov 94, pp 17 - 27.

[5] D. Abramson and J. Giddy, "Scheduling Large Parametric Modelling Experiments on a Distributed Meta-computer", PCW '97, September 25 and 26, 1997, Australian National University, Canberra, pp P2-H-1 – P2-H-8.

[6] D. Abramson, I. Foster, J. Giddy, A. Lewis, R. Sosic, R. Sutherst, N. White, The Nimrod Computational Workbench: A Case Study in Desktop Metacomputing, Australian Computer Science Conference (ACSC 97), Macquarie University, Sydney, Feb 1997, pp 17 - 26.

[7] D. Abramson, R. Sosic, J. Giddy and B. Hall, "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[8] D. Becker, T Sterling, D Savarese, J. Dorband, U Ranawak, C. Packer, "BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION", Proceedings, International Conference on Parallel Processing, 95.

[9] D.Waitzman, C. Partridge and S. Deering. Distance vector multicast routing protocol. *RFC 1075*, November 1988.

[10] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations, *Inform. Process. Lett.*, 11: 1-4, August 1980.

[11] http://beowulf.gsfc.nasa.gov/

[12] http://www.activetools.com

[13] http://www.mil3.com

[14] http://www.myri.com/

[15] I. Foster, C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit. ", Intl J. Supercomputer Applications, 11(2):115-128, 1997.

[16] J. Moy. Multicast extension to OSPF. *Internet Draft 1584*, 1994

[17] K. Power, B. Qiu, J. Breen and M. Hassan. A Multicast routing protocol for ad hoc networks, *APCC/ICCS '98*, November 1998.

[18] K. Power, B. Qui and J. Breen. AMP: A multicast routing protocol for ad hoc networks, *ISPACS '98*, November 1998.

[19] Lewis, D. Abramson., R. Sosic, J Giddy, "Tool-based Parameterisation : An Application Perspective", Computational Techniques and Applications Conference, Melbourne, July 1995.

[20] M. Behzad. *Graphs and Digraphs*. Prindle, Weber and Schmidt, 1979.

[21] M. Parsa and J. J Garcia-Luna-Aceves. A protocol for scalable loop-free multicast routing. *IEEE JSAC*, 15:316-331, April 1997.

[22] M. S. Corson and S. G. Batsell. A reservation-based multicast routing protocol for mobile networks: Overview of initial route construction. *Wireless Networks*, 1:427-450, November 1995.

[23] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, W. Su "MYRINET: A GIGABIT PER SECOND LOCAL AREA NETWORK", IEEE-Micro,Vol.15, No.1, February 1995, pp.29-36.

[24] R. Bhattacharya and A. Ephremides. A distributed multicast routing protocol for ad hoc (flat) mobile wireless networks. *8th Int. Sym. Personal, Indoor and Mobile Wireless Communications*. Pages 877-881.

[25] S. E. Deering et al. The PIM architecture for wide-area multicast routing. *IEEE/ACM Trans. Networking*, 4:153-162, April 1996.

CHAPTER 22

Cluster and Grid Infrastructure for Computational Chemistry and Biochemistry

KIM K. BALDRIDGE, WIBKE SUDHOLT, JERRY P. GREENBERG, CELINE AMOREIRA, YOHANN POTIER, ILKAY ALTINTAS, ADAM BIRNBAUM, DAVID ABRAMSON, COLIN ENTICOTT, and SLAVISA GARIC

Many computational chemists requiring significant and relatively flexible resources have turned to parallel clusters to solve increasingly complex problems. Evolving hardware technology and grid resources present new opportunities for chemistry and biology, yet introduce new complexity related to grid, web, and computational difficulties. Here, we describe our experience in using the GAMESS quantum chemistry program on clusters and our utilization of evolving portal, grid, and workflow technologies to solve problems that cannot be solved on individual machines.

22.1 INTRODUCTION

Computational chemists tend to require multiple types of facilities based on the rather large diversity of chemical problems studied, and the interoperability of resources sought, to carry out increasingly complex computational science experiments. The complexity of the associated computational chemistry software forces chemists to focus on advanced high-performance computing methodologies to exploit rapidly changing architectures. In particular, many groups have long recognized the importance of parallel computers in increasing the range of chemical systems that can be treated by quantum mechanical techniques. In the computational chemistry community, for example, a code called *GAMESS* (general atomic and molecular electronic structure system) [1, 2] has been the focus of many years of community code development. GAMESS has evolved over the years into a highly reputable code

Parallel Computing for Bioinformatics and Computational Biology, Edited by Albert Y. Zomaya
Copyright © 2006 John Wiley & Sons, Inc.

for accurate representations of molecular structure and properties through quantum mechanical techniques. In addition, much effort has gone into providing optimal performance on cutting-edge, high-end architectures, and more recently, it has been central to the development of robust and user-friendly working environments.

On the basis of our experiences from a hard science background and a significant information technology (IT) depth, we believe that the effective use of such infrastructures is limited by (1) rapid pace of technological change in IT infrastructure, which requires users to constantly learn new terminology and tools, (2) lack of flexibility from system components that inhibit exploratory analysis by the end users, (3) IT-oriented abstractions that are difficult for domain scientists to work with, and (4) communication optimization between the computer science, physical science, and life science communities.

Computational infrastructure technology has been rapidly evolving over the past few years: Hardware resources have been moving away from traditional high-end supercomputers and turning toward commodity clusters, remote grid computing, and even desktop grid systems. In addition, new large-scale data-management systems and services-based software infrastructures are significantly enabling the management of software on such architectures. User portals, applications, and environments in which end users directly interact have been effective at shielding the end user from the changing technologies by providing a domain-specific layer, which provides stable interfaces for users, while adapting to new technologies under covers.

What we describe in this contribution are our experiences as we have evolved the GAMESS computational chemistry community of users through new opportunities offered by the cluster and grid revolution, including (1) standard running of computations on simple cluster computers, including the evolution in node quality, (2) the use of portal technology for ease of execution on cluster architectures, (3) the employment of efficient grid-execution systems for large-scale computing, and (4) exploitation of middleware technology for running complex computational experiments.

22.2 GAMESS EXECUTION ON CLUSTERS

In the GAMESS code, parallelism is programmed through the distributed data interface (DDI) [3] layer. Beneath the DDI layer are calls to sockets, MPI [4], or shared memory. On Linux clusters, GAMESS typically uses sockets for communication between the processors. A separate program starts two GAMESS processes on each host, which then establish socket communication between themselves. The "compute processes" do almost all the work. The "data server" processes control shared memory distributed over all the hosts and store and retrieve data when requested by the compute processes. At present, only a limited number of higher order quantum chemistry computational methods use the distributed memory facilities. All GAMESS processes run on the compute nodes. The "master" node (node 0) handles all user input and output tasks, in addition to contributing to the computational tasks that all other compute server processes perform.

The conventional means of running GAMESS on clusters has been to log on to a cluster front end directly and to run shell scripts provided for GAMESS installation and execution. Typically, the shell scripts are tailored to the particular platform being used, with variations in such system-dependent considerations as parallel APIs (e.g., sockets, MPI) and queuing systems (PBS [5] and SGE [6]). This poses somewhat of a burden on users who want to solve their computational chemistry problems of interest and are not necessarily concerned with system architecture. Subsequently, we will describe our efforts to provide a layer of abstraction for users from the computational resource.

Another difficulty facing users and cluster managers is the system environment, which may vary widely between clusters. For a cluster with N hosts, there are N copies of the operating system to manage. Furthermore, each host must be updated with items such as new software, security patches, and account information. If the system is not maintained with a consistent set of software on each host, it may become unstable and endanger long running calculations. The ROCKS cluster-management system [7] provides a means to deploy a cluster in a short period of time and to maintain consistency between all the nodes. ROCKS provides not only a solution for maintaining and updating Linux clusters but also provides a common interface for the user regardless of cluster hardware. In terms of a GAMESS installation, deployment to a ROCKS cluster involves minimal, if any, reconfiguration.

GAMESS performance is quite good on cluster architectures, providing scaling close to linearity for large molecular systems. Quantum codes, in general, have many different task types, which are invoked depending on the specifics of the requested computation. These tasks range from being highly serial to fairly parallel, as well as requiring anywhere from moderate to extreme memory and disk resources. Thus, various parts of the code will have different levels of performance characteristics. An example would be the comparison of the matrix multiplication type steps to the calculation/processing of the atomic orbital integrals. As such, the move to parallel cluster architectures has been very beneficial to the performance of GAMESS, and through careful algorithmic enhancements such as those discussed earlier, we have been able to see significant scaling improvements.

Figure 22.1*a* shows the performance of GAMESS across a wide variety of platforms, from tightly coupled supercomputers such as SDSC-Blue Horizon to the large shared-memory SDSC-SUNHPC platform, to the TERA-InCore platform, and to clusters of computers (Meteor = Pentium III based and Compaq = Alpha based). In fact, the Alpha cluster is one of the better platforms in terms of overall performance, as it retains high efficiencies (>95%) with increasing nodes. We should also note that the TERA performance is artificially high because of the incompleteness of the computation that is enabled in GAMESS for that platform. In fact, the clusters are, in general, quite competitive to the large supercomputer infrastructures, with essentially linear performance even on a small test case involving the luciferin molecule with the chemical formula $C_{11}N_2S_2O_3H_8$ and no molecular symmetry. The total memory required for processing of atomic orbital integrals in this case is 3.8 GB. In general, one should expect a faster reduction in performance for distributed platforms when

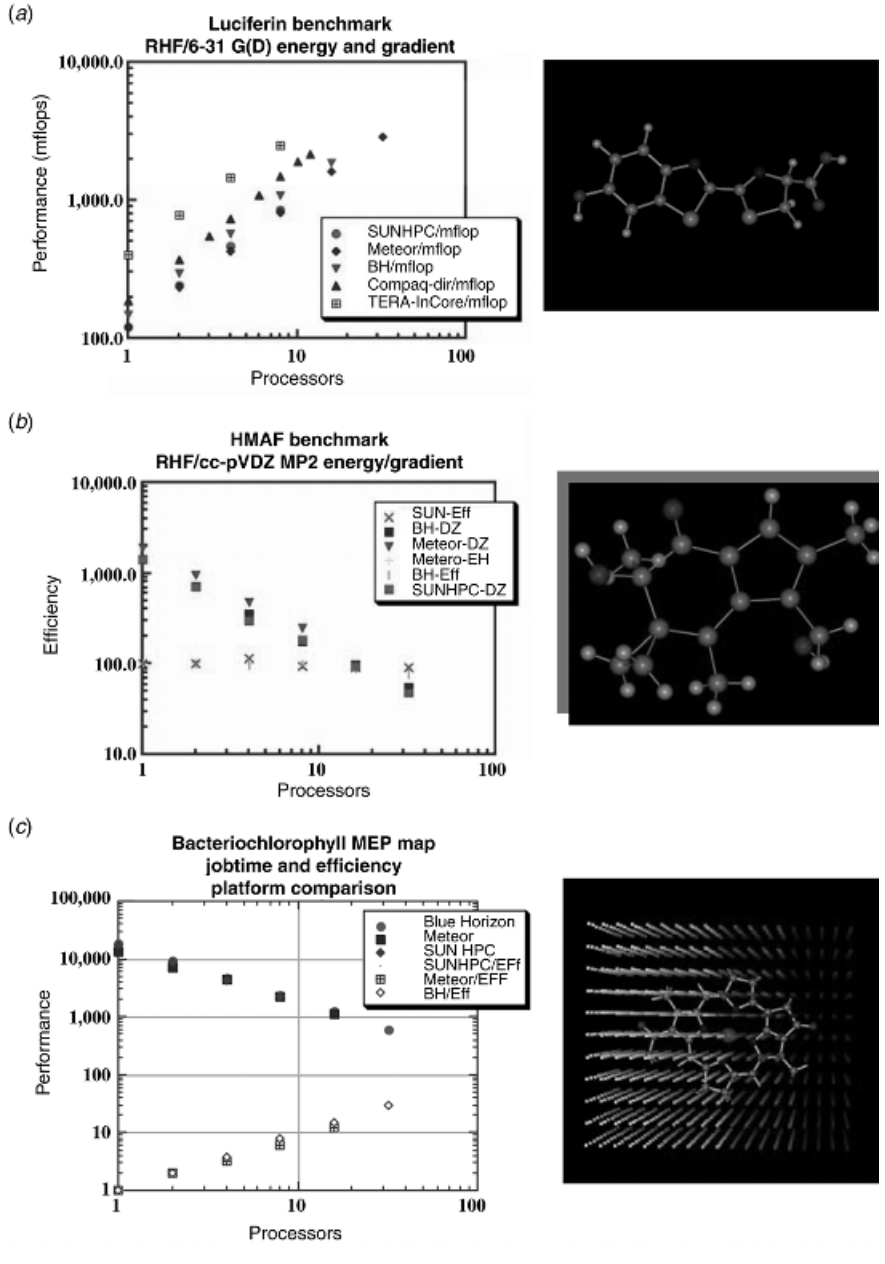
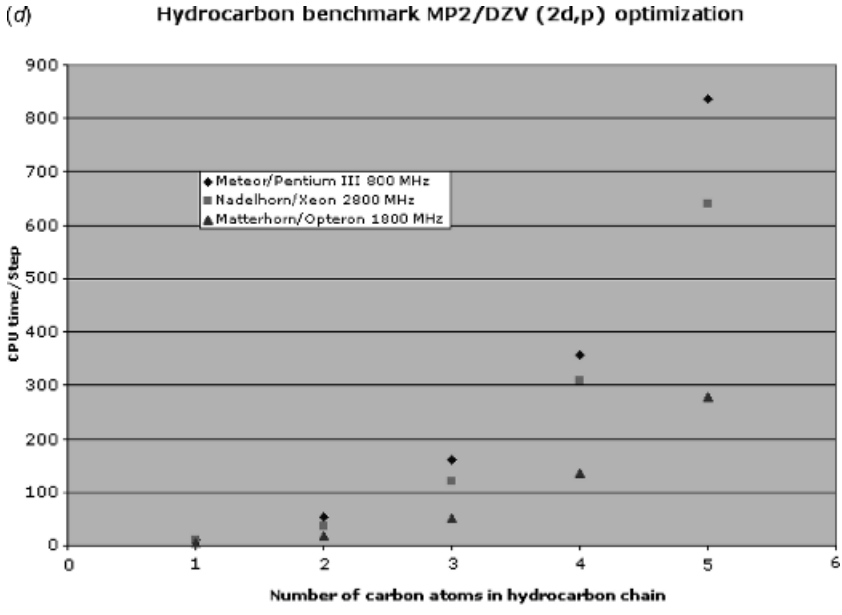


Figure 22.1 (*a*) Performance vs. number of processors for an RHF energy and gradient calculation on luciferin. (*b*) Efficiency vs. number of processors for an MP2 energy and gradient calculation on hydroxy-methyl fulvene. (*c*) Performance vs. number of processors for an electrostatic map calculation on bacteriochlorophyll. (*d*) CPU time per optimization step for an MP2 optimization of hydrocarbon molecules of varying lengths for two processors. (*e*) CPU time vs. number of processors for the optimization of a particular hydrocarbon (butane).

(*d*)



(*e*)

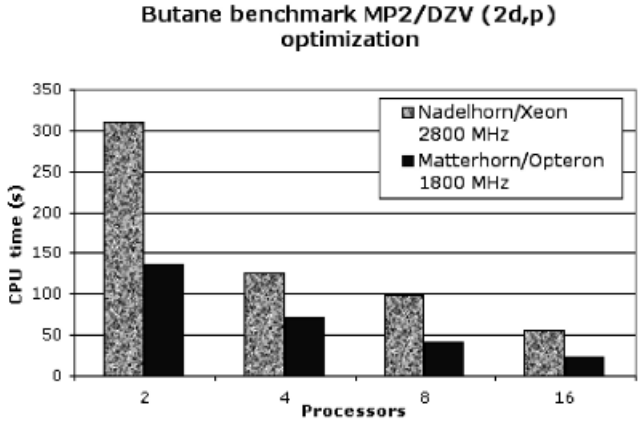


Figure 22.1 *Continued.*

tasks run out (e.g., computations involving small molecular systems on large numbers of nodes).

A particularly good example of scalability occurs with a rather CPU and memory intensive capability of GAMESS, involving the dynamic correlation method — second-order Møller–Plesset (MP2) theory. In this case, our scalable distributed-data algorithms target the increasingly large aggregate memories available with cluster architectures. The design of the distributed-data algorithms is enhanced considerably by the availability of one-sided forms of communication that facilitate remote data

access in a shared-memory style of environment. This has shown to be particularly important in methods such as MP2. A distributed-data second-order MP2 energy algorithm is now available, which typically maintains an efficiency of over 90% with increased problem size. Figure 22.1*b* illustrates the performance for a relatively modest calculation involving an important cancer drug target, hydroxy-methyl fulvene (17 heavy atoms). Here, we show MFLOPS (solid) and efficiencies (cross-lines) again on three different architecture types. This computation retains more than 90% efficiency with increasing node application on the Pentium III cluster (triangles) on which it was run.

We have also investigated the computational performance of molecular property computations, which typically involve the calculation and handling of potentially very large N^3 grid points. The time to calculate 3D surfaces can be a significant, or even the major, portion of the calculation when computed on a single node. When the calculations are parallelized, the 3D serial grid calculations become much more significant relative to the parallel calculations. All these effects are multiplied, when several levels of grid properties have to be computed at once (e.g., 10 or more grids per molecular system). For example, on a R12000 processor, the time to compute one $60 \times 60 \times 60$ grid exceeds 1 h (and this is a very small grid size).

To improve efficiency on clusters, we have divided the grid into groups of processors and accumulated the results via the DDI layer. Figure 22.1*c* illustrates the overall performance in terms of CPU and efficiency measures for the computation involving the moderately large bacteriochlorophyll molecule electrostatic field.

Finally, we have the opportunity to make a few comparisons due to enhancements in processor speed as we have now cluster systems with Pentium III processors, Xeon processors, and Opteron processors. Here, we have performed several benchmark computations on hydrocarbons with increasing number of carbons (for comparative purposes, we consider all-staggered forms of the hydrocarbons here). Figure 22.1*d* shows the CPU time (in seconds) per optimization step as a function of hydrocarbon length across these three platform types. The processor speeds for the three platforms are shown. The Opteron cluster shows the best performance. It is important to keep in mind that there are other considerations to the overall speed of the calculations other than just the clock speed on the particular processor. Figure 22.1*e* shows the performance of a geometry optimization calculation for a single hydrocarbon across a sequence of node combinations. This illustrates the difference in performance particularly for the Xeon vs. the Opteron cluster type. For this small molecular system, the efficiency is optimal up to 16 nodes, at which point the workload is no longer an issue.

Although it is certainly the case that GAMESS performance is enhanced by the continued increase in processor speed, processors on the slower side can also compete in total performance by executing multi-tasking, which has been enabled by the compiler. In addition to the importance of speed to main memory, the capacity of the main memory, the speed of access to the disk drives, and the latency and bandwidth on the network controllers all contribute to the overall performance observed for GAMESS jobs.

22.3 PORTAL TECHNOLOGY

Our first attempt at providing a means to shield users from the intricacies of parallel computational systems involved several schemes for submitting GAMESS jobs to remote resources and retrieving the results through the *QMView* visualization software [8]. We also developed tools such as a cluster submission/monitoring program, which allows users to choose nodes and submit jobs via an X11 interface. These were primarily for "proof of concept" and demonstration purposes; they were never established as production "services." All of those methods were originally designed before the advent of the world wide web and are now essentially obsolete, superseded by new grid technologies.

Subsequently, the SDSC portals group created the GridPort Toolkit [9], which provided portal developers with a Perl API for establishing secure web portals capable of submitting, controlling, and gathering the output of jobs to remote high-performance computing resources. The GridPort API also includes file transfer capabilities that a portal developer can use to transfer files such as input and output data using GridFTP or the storage resource broker (SRB). To access a GridPort Portal like the *GAMESS Portal* [10], a user logs on to a website, selects input files, and chooses a remote platform to run their jobs on. The portal provides information on job status and stores the output on the SDSC SRB system [11]. GridPort uses the Globus Toolkit [12] to handle authentication, security, and job submission and control. Figure 22.2 gives a flowchart of jobs and data through a portal.

In our initial implementations of the GAMESS Portal, the "storage device" used to archive the simulation results was an NFS-exported file system. In the updated system, all data files are "collections" within the SRB. Access to data stored on the SRB is provided by web portals, which allow a user to download and upload files by SRB client tools on UNIX systems using the "srbBrowser" program on Windows systems, or via the SRB website. We are also building infrastructure for computational chemistry database technologies to be included in the environment.

Although GridPort has proven to be a useful tool and is widely utilized, it lacks a basis on which to build complex and dynamically reconfigurable workflows. Although complexity is hidden from the end user, it is not hidden from the application scientist or portal developer: Many CGI scripts and HTML documents have to be created or copied and modified from existing portals. More complications come in when new computational platforms are added, since all runtime details on the new platform must be added, including the queuing mechanism and location of the executable and directory structures. What is lacking is an overall architecture that provides the user and the application programmer with an interface that shields them from the complexity of the HPC resources available and allows them to concentrate on solving their scientific problem, while allowing scientific programmers to integrate their software into the system in a simple and effective manner. The next generation of portal technology provides a further layer of abstraction for the applications programmer and exploits the latest grid technologies.

The web and grid technologies briefly described earlier can be used to run GAMESS jobs without making changes in the GAMESS program itself nor in the

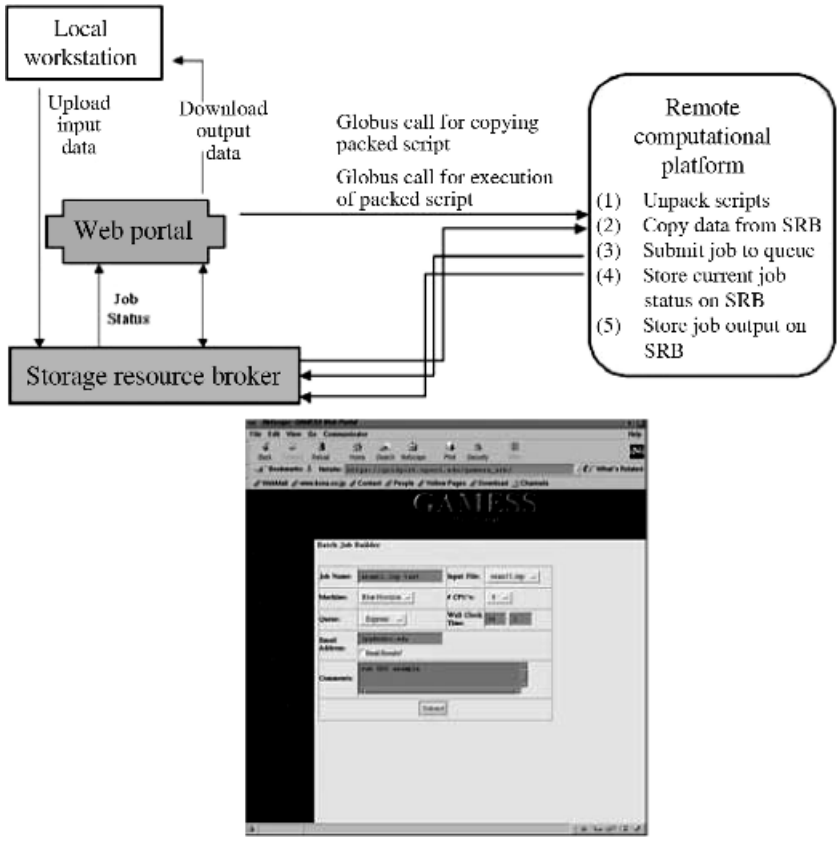


Figure 22.2 Flowchart illustrating the operation of the GAMESS Portal.

mechanism of deployment. Work is currently in progress to enable GAMESS to output structured data in the form of extensible markup language (XML) documents and JAVA objects based on the GAMESS schema [13–15]. This will facilitate data transfer between programs in a workflow and also provide a basis for a database structure (Section 22.5).

22.4 RUNNING GAMESS WITH NIMROD GRID-ENABLING INFRASTRUCTURE

Over the last few years, combinations of computers, called "grids," have been developed, which couple geographically distributed resources such as supercomputers, workstations, clusters, and scientific instruments [16, 17]. Newly constituted grid organizations such as the U.S.-based *TeraGrid* [18] and the Swiss-based *SwissBioGrid* [19] have begun to provide the infrastructure to support global and interdisciplinary

collaboration in science and engineering in ways that were not previously possible. They enable a new discipline called "*e-Science*." In addition, because of involvement in various projects, computational researchers may even build their own "private" grids when they combine access to several computers at different institutions worldwide.

Unlike traditional high-performance computing systems, grids provide more than just computing power, they provide a new "Cyberinfrastructure." Computational grids address issues of long-distance networking, distributed storage, wide-area scheduling, and resource discovery in ways that allow many resources (e.g., computers and scientific instruments) to be assembled, on demand, to solve large, complex problems. Grid applications potentially allow real-time processing of data streams from scientific instruments such as synchrotrons and telescopes in ways that are much more flexible and powerful than those currently available. Through the pooling of resources, grids provide significant and flexibly adjustable computational power, which can be exploited by quantum chemistry and other compute-intense calculations.

Common challenges in establishing a grid infrastructure present the diversity of hardware and software, restrictions in the network connections, different site policies, and communication difficulties between the various institutions and disciplines involved. In addition, scientific program codes are often complex and were developed over many years by many people, with specific applications and efficiency in mind. As exemplified in Section 22.2 for the quantum chemical package GAMESS [1, 2], selected codes have been adapted to run in parallel on various compute cluster and supercomputer systems. Despite this portability, the adaptation to loosely connected and heterogeneous grid environments is challenging. Although clusters, with similar hardware and fast, reliable network backbone between the nodes, make parallel computations very efficient, geographically distributed grids suffer from high network latency, uncontrollable downtimes, and diverse hardware. This makes it currently less desirable to run a single job in parallel on different grid machines. Therefore, we concentrate less on the specifics of the optimization of the application codes for parallel usage on grids at the moment, and instead, we focus here on connecting the compute clusters to distribute many independent jobs onto a personal "grid queuing system." As we have seen, even this much-needed capability is still a challenging mission.

An obvious way of using such widely distributed grid resources is when a single "experiment" requires processing multiple different input scenarios, as this is a typical embarrassingly parallel task. We have initially undertaken to provide such a capability with two separate strategies, each having their advantages and implications for computational chemistry applications: One mechanism involves calling GAMESS within the Nimrod Toolkit [20] for distributed parametric modeling, which we will discuss in this section. The other mechanism involves calling GAMESS within a desktop grid environment, which will not be covered here (for more details, see Ref. [21]).

The experimental or computational scanning of a parameter space and the search for optimal solutions to construct or test hypotheses or algorithms is a common and fundamental task in scientific research. Although the generation of such jobs sounds relatively easy, the computational requirements and the amount of bookkeeping steeply rise with the number of generated jobs. Furthermore, it is almost impossible

for application scientists to adjust the load of their calculations on various resources in an adequate and efficient manner manually. However, as all jobs are basically similar to each other — they only differ in the values of certain parameters — and they are not directly coupled to each other, this is a perfect task to be solved by grid computing.

We used the Nimrod family of tools for distributed parametric modeling [22–24] to construct, launch, control, and gather a quantum chemical parameter scan with GAMESS. Developed at the Distributed Systems Technology Centre at Monash University, Australia, *Nimrod/G* [22, 23] manages the execution of studies with varying parameters across distributed computers. This software responsibility for the overall management of an experiment as well as the low-level issues of distributing files to remote systems, performing the remote computations, and gathering the results. When users describe an experiment to Nimrod/G, a declarative "plan file" is developed, which describes the parameters, their default values, and the commands needed to perform the work. Apart from this high-level description, users are freed from much of the complexity of the grid. As a result, Nimrod/G has been very popular among application scientists.

Nimrod/O [24] is a variant of Nimrod/G, which performs a guided search of the design space rather than exploring all combinations. Nimrod/O allows users to phrase questions such as: "What set of design parameters will minimize (or maximize) the output of my model?" If the model computes metrics such as cost and lifetime, it is then possible to perform automatic optimal design. A commercial version of Nimrod, called *EnFuzion*, has also been produced. A web portal, called the *Nimrod Portal*, provides access to Nimrod via conventional browser technology. Of the different modes of user interfaces offered by the Nimrod software, we utilized the portal interface in this work.

Nimrod is built on a variety of middleware layers, most importantly Globus [12], through which it can, for example, interact with remote cluster queuing systems. Considering constraints such as time or cost, Nimrod dynamically detects suitable resources from a user-provided list of grid-enabled computers, modifies and ships the corresponding input files to those resources, submits the individual jobs via "agents" scheduled to run on the machines, and collects the results, while handling events such as node and network failures. Nimrod's core is a relational database, which stores the experimental details.

In our case, we prepared, executed, controlled, and gathered GAMESS calculations with the help of the Nimrod Toolkit. As an application, we used the GAMESS/Nimrod setup to parameterize a so-called "group difference potential" (GDP) [25, 26]. This is a quantum chemical pseudopotential designed to be used in future research, to cap atoms at carbon–carbon single bonds cut in hybrid quantum mechanics–molecular mechanics (QM/MM) calculations or to allow free energy difference determinations between fluorine and methyl groups. The former method enables accurate yet fast modeling of chemical reactions in realistic, yet large and complex molecular systems such as solutions, biomolecules, and materials, by subdividing the system into an active QM and a surrounding MM parts. The latter allows estimation of the effects of functional group substitutions in derivatives of the original compounds, for example, in drug or catalyst development.

Our potential function consists of four adjustable parameters, and the parameterization is carried out using quantum chemical calculations involving the GAMESS package, in this case, for the small prototype molecule ethane. The integrity of each parameter combination is measured by a weighted least-square difference cost function, which compares various properties of the original molecule with the pseudomolecule. First, we scanned subsequent regions of the parameter space in their entirety, applying different types and densities of point spacing, using the Nimrod/G sweeping tool to identify the most interesting region for our purposes. Secondly, we performed cost function minimizations on this subspace with the Nimrod/O parameter optimization tool.

The parameter sweeps resulted in experiment sizes of up to about $60{,}000 \times 4$ single GAMESS calculations. Performing this number of tasks manually on a single machine is practically impossible, but with the Nimrod technologies lasted only a few days when distributed automatically on a grid of computers. The actual experiments were mainly performed during the Fourth Pacific Rim Applications and Grid Middleware Assembly (PRAGMA) workshop and the Supercomputing (SC) 2003 conference. For these events, we were able, with the help of the PRAGMA collaboration [27], to accumulate the necessary number of resources to form a huge grid testbed, spanning a range of hardware architectures, software setups, organizations, and countries. Many of these machines were compute clusters running Linux, often under ROCKS configuration management. Analysis of the resource usage shows the efficient, dynamic, and flexible distribution of jobs over the grid testbed to minimize the overall execution time for each experiment. Considering the enormous success of such experiments, there was one considerable obstacle, involving misconfigured Globus installations on a few of the machines, which again shows the difficulties in dealing with the grid middleware directly.

At the end of the cost function minimizations, we obtained an optimal parameter set for our ethane GDP. Currently, this result is tested under different environmental conditions, for larger molecules, and with other quantum chemical methods. Up to now, the outcomes are encouraging and the pseudopotential seems to be stable under a variety of circumstances. More details of the GAMESS/Nimrod experiments can be found in Refs. [25, 26]; further publications are in preparation.

The success of this project encouraged us to apply Nimrod also in other fields of our research. Of particular interest in our current efforts is the behavior of ligands in proteins, but similar principles can also be used, for example, to study the driving forces involved in protein–protein interactions. Both topics are of invaluable importance for the understanding of biological processes and the design of pharmaceutical agents.

In recent years, we developed techniques that simplify the modeling of the protein environmental influence on a ligand via a combination of quantum chemical and continuum electrostatics approaches. Our current work targets the automation of a molecular docking procedure, which is typically known to be computationally challenging both from the perspective of computational resources and computational accuracy. Such calculations are difficult because of the fact that one would like to investigate all possible configurations in the ligand–protein interaction space to find

their best relative position. However, the correct representation of electrostatic and non-electrostatic energies as well as entropic contributions in the overall scoring cost function is not fully established. We target both the configurational as well as the energetic surface analysis, since the optimal protein–ligand interaction is not determined solely by one or the other but by a complex interplay between energetic stabilization and configuration selection. The initial parameterization will be done on systems for which the real positions of ligand and protein are known from experimental results in the Protein Data Bank (PDB) [28].

Tools such as Nimrod greatly facilitate carrying out procedures such as described, as they can manage the rather large set of individual calculations that one would like to do to obtain the most accurate and thorough results. Furthermore, apart from existing parameter sweeping and objective parameter optimization facilities in Nimrod/G and Nimrod/O, recently an interactive subjective optimization algorithm was implemented into the Nimrod Toolkit (*Nimrod/OI*). An objective cost function can thus be applied to monitor the energetics associated with the protein–ligand interaction in any particular molecular configuration of the ligand docked into the protein, and a subjective cost function to monitor the steric and overall configuration details in terms of special parameters. The subjective optimization procedure needs direct user intervention and decisions, and therefore, requires the use of a visualization tool, for which we plan to invoke our QMView [8] molecular computation and analysis tool.

As such, our particular docking protocol requires executing two major computational chemistry or biophysics packages: The GAMESS software, which was discussed earlier, has just to be the invoked once for each ligand structure, to obtain its hydrogen positions and atomic charge distribution. Then, the adaptive Poisson–Boltzmann solver (APBS) [29, 30], a code for biomolecular dielectric continuum solvation calculations, computes the electrostatic energies of protein, ligand, and protein–ligand complex in solution for each of their relevant positions, from which the corresponding binding energies can be determined. However, particularly due to the complexity of protein structures, a variety of auxiliary tools is also required, to prepare, manipulate, and analyze the docking calculations such as those to access the protein data, to set up the GAMESS and APBS input files, to bring the quantum chemical results into the right format, to assign the molecular force field, to estimate non-electrostatic contributions, and to visualize the results. A principal methodology to automate this overall procedure will be introduced in the next chapter.

22.5 COMPUTATIONAL CHEMISTRY WORKFLOW ENVIRONMENTS

Although Nimrod plan files allow the user to construct basic pipelines, there is a considerable interest in the ability to combine more than one computational code, to build flexible, complex, and reusable workflows for high-throughput studies on significant numbers of molecules or with a variety of methods. Recently, we have begun to develop a "computational chemistry prototyping environment," which would enable researchers to design computational experiments which span multiple computational and analytical models, and in the process, to store, access, transfer, and

query information. This requires the integration of a variety of computational tools, including the chemistry software, preparation, visualization, and analysis toolkits, as well as database programs. The overall infrastructure is being referred to as the "Resurgence" RESearch sURGe ENabled by CyberinfrastructurE project.

We are realizing this with the help of the *Kepler* collaborative system for scientific workflows [31–33], which is based on dataflow process networks. Exploiting Kepler addresses various levels of complexity, detail, and control desired by different factions of the rather diverse computational chemistry community. A major focus here is not to provide predefined workflows for pure application, but a set of highly modular and extensible elementary and composite workflow pieces, which by individual combination allow researchers to explore new paradigms for their science. Such explorations particularly include new ideas in complex molecular modeling, such as exist in the biological or material sciences. For example, we would like to refer here to our efforts in protein–ligand docking, described in detail in Section 22.4.

Underlying Kepler is the Java-based *Ptolemy II* system [34] for heterogeneous, concurrent modeling and design. It also provides the foundation for the graphical user interface (GUI) called *Vergil*, which facilitates the setup and execution of complex workflow scenarios with a convenient workflow editor and libraries of component workflow tools. Dragging and dropping a number of predefined so-called "actors" into the workspace construct a functional model, which comprises a complete chemistry computational process. These are then connected together between their input and output "ports," defining the routes for the transfer of data or the triggering of other actors. Certain "parameters" of the actors can also be modified after double clicking. Examples of fundamental Kepler actors for computing are command-line execution, Globus jobs, GridFTP, proxy certificate management, and web service consumers. Several actors may be combined to build more complex "composite" actors, which allows defining hierarchies. Finally, a so-called "director" for each workflow part is used to steer execution of the merged actors and to manage scheduling, dispatching threads, running of code, and similar tasks. The created workflow is saved in a Ptolemy II-specific XML dialect called modeling markup language (MoML). For the GUI and an example computational chemistry workflow, see Figure 22.3.

Overall, the software under development can be described as consisting of three different layers: (1) the "control layer" on the top guarantees communication between different molecular modeling applications and databases, (2) the "translation layer" in the middle is responsible for constructing the inputs and analyzing the outputs of each program involved, and (3) the "execution layer" on the bottom provides the connection to the computational resources, the transfer of files between machines, and the running of individual jobs.

Following the previous success, the dynamic grid distribution facilities on the execution layer are realized by Nimrod/G, as described in Section 22.4. Combined with the static grid-execution facilities of Globus, we believe that this will give scientists a tool that has the ability to catapult computational chemistry research to new heights. In such a complex workflow, it is possible to invoke a Nimrod run as just one component of a series of calculations. From the corresponding list of input files and a template, a plan file is automatically constructed. The template is specific for

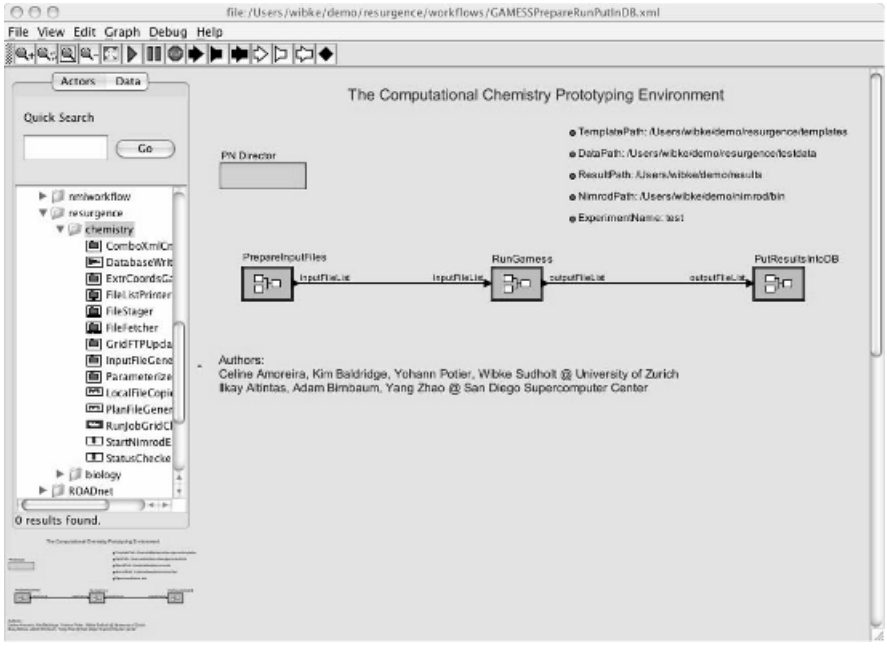


Figure 22.3 Example workflow of the computational chemistry prototyping environment in the Kepler GUI, developed within the Resurgence project.

each computational code or installation. Currently, this setup works for the GAMESS quantum chemical program package [1, 2]. The corresponding experiment is then added to the Nimrod database. The same applies to the grid resources, which the user selects prior to running the workflow. Finally, the computational experiment is started and monitored until execution stops. At the end, the list of generated output files is given back. Although Nimrod is currently accessed via its command-line API, such a service could also be implemented by adding a web service interface for Nimrod. In both ways, Kepler will be able to invoke parameter scans and searches, and the results can be processed and passed to later computations.

The translation and control layers are more difficult to realize. Here, the diversity of the computational chemistry software represents a considerable issue. Although the codes mostly read or write text or binary input and output files, these generally have proprietary file formats, which are incompatible to each other. In addition, not all corresponding sources are open or they are hard to adapt due to their grown complexity. Thus, researchers usually apply a variety of transformation procedures in between two programs to connect these. Either these are self-written scripts, freely available molecular file format conversion tools such as OpenBabel [35], or the two codes involved are directly called from one another.

To solve this very general problem, two aspects are of importance: First, the input and output files should be in an easier format to process, which allows both automatic

and human reading and writing. For this, XML [36] seems to be a perfect choice. Secondly, a common data format may also be desirable. An attempt into this direction is the chemical markup language (CML) [37, 38] or its recently started computational chemistry derivate CMLComp. Unfortunately, experience shows that it is very difficult to define one overall data format for a complex field such as molecular modeling, and we ultimately even want to connect codes across disciplines. In addition, ontologies should be part of the research and not part of the file format. Therefore, we believe it is best for now to provide the user with code-specific XML, so that he or she does not need to think about the file formats any more, but only about the correct mapping of data. This means that the individual calculation programs need to be internally or externally (e.g., by XSLT) wrapped to be able to communicate with each other (Fig. 22.4). We have already performed a corresponding integration for the GAMESS quantum chemistry package inside the code (Section 22.3) and within the Resurgence project.

When one considers high-throughput calculations, another important issue is the interfacing to databases. On one hand, only databases provide access to the large numbers of molecular geometries needed as input for the computational chemistry calculations. Thus, we plan to interface our workflow system to experimental structure databases such as the RCSB PDB [28] and the CCDC Cambridge Structural Database [39]. On the other hand, the outcomes need to be stored in databases as well, as only these allow managing the huge amounts of newly computed data. In addition, the detailed and systematic storage permits to perform data mining on the results to find relationships between properties that were not known before and could also not easily be deduced from small numbers of calculations.

Therefore, we are currently building a relational databank to store the XML output of GAMESS jobs, named "QM-DB." When it is based, for example, on the professional IBM DB2 database system [40], the storage of data could work by XML-relational mapping defined in a document access definition file and processing of the results via a corresponding stored procedure. Our overall view is that — to guarantee the integrity and accessibility of the data — there should be only one open worldwide GAMESS data repository, similar in spirit to the PDB. In addition, analogous layouts could be introduced for other molecular modeling codes. Federation may then combine all these databases. This strategy is parallel to our ideas on XML and CML.

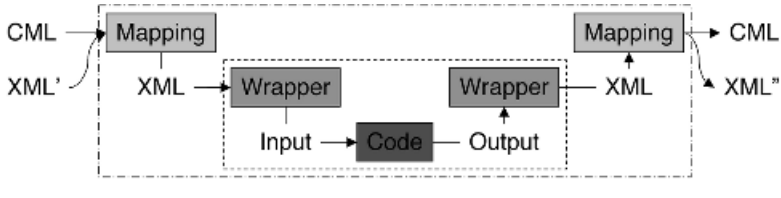


Figure 22.4 Wrapping of a computational chemistry code to read and write XML and mapping of the XML information for connection with other codes.

We are currently implementing some of these features into the Resurgence add-on module [41] for the Kepler workflow collaboration. A semi-automatic installation procedure might ease the setup of the large number of different necessary software tools (Ptolemy II, Kepler, Resurgence, Python, PostgreSQL, and Nimrod/G). Nevertheless, even though MoML-defined workflows can be executed independently from the GUI, the overall system is inherently coupled to a central machine where Kepler and Nimrod need to be installed. In the age of grid computing, however, ultimately a more flexible and dynamically reconfigurable design is desirable. Thus, the next generation of workflow architecture, constructed under the impetus of the newly interdisciplinary chemical and biological outlook, will be more generally layered (e.g., an interface layer, a middleware layer, and a resource layer), shielding users even more from the complexity of the underlying grid system. The associated tool, INFORMNET [42], developed in parallel to our Kepler efforts by Stephen Mock et al. at the San Diego Supercomputer Center, provides convenient mechanisms of abstraction of functionality, interface, hardware, and software, tested in an application-rich environment.

22.6 CONCLUSIONS

Calculations that predict the structures, energetics, and other properties of experimentally known or unknown molecules are a fundamental resource in modern chemical and biological research. Understanding molecular systems at a variety of levels of detail and accuracy requires a variety of different types of computational capabilities to be accessible. The goal is nevertheless the solution of problems of general chemical and biological interest, especially for the experimental side — how molecular and electronic forces determine the details of structures and the dynamics and energetics of chemical reactions.

In the past 5 years, the quantum chemistry program package GAMESS [1, 2] has served as a strategic application for prototyping advances in cluster and grid architecture and connectivity. First of all, sophisticated parallelization approaches and interfaces allow running GAMESS jobs very efficiently on clusters and supercomputers. Together with the steady advances in hardware, this enables researchers to study increasingly larger systems of chemical and biological interest in more detail and with greater accuracy. A GAMESS GridPort web portal at SDSC permits job submissions on a variety of platforms, most predominately clusters, using the technologies of the SRB and Globus middleware tools to assemble, carry out, and monitor jobs and store the results. This shields the complexity of the grid middleware from the end user (although not from the application developer) and eases the calculations by client-independent Internet access and flexible server selection.

The execution of GAMESS or other experiments via the Nimrod Toolkit permits the easy setup and distribution of parameters sweeps and optimizations on computational grids. Also here, the scientist does not have to deal directly with the middleware software and, in addition, does not need to worry about the allocation of jobs to machines, which is done automatically and in a high-throughput manner, opening

new possibilities for research. Desktop grids provide an alternative approach based on different platforms and with different targets for application, but similar advantages. Finally, the complex combination of diverse computational chemistry — and later biology — tasks and the grid distribution of the resource-intensive calculations can be realized by the scientific workflow systems currently under development. Here, the goal is to flexibly integrate a variety of computational tools including molecular calculation software such as GAMESS, visualization and analysis toolkits, and database programs, thereby providing reusable building blocks.

This advanced set of cluster, portal, grid, and workflow infrastructure is now enabling considerable progress toward the integration of chemical and biochemical calculations across all scales from molecules to cells to organisms, with the quantum chemical GAMESS application providing the basis for a more generalized chemistry and biology framework. Examples for this are the GridPort and Nimrod integrations of APBS and further scientific software. The objective is to put experimental data and computational modeling within reach of users in a variety of disciplines through intuitive, configurable, and easy-to-use graphical and web interfaces, but also to still support all possibilities for experts in each field.

Our ultimate goal is to create a software environment that is able to run applications from any scientific domain with little configuration effort by users or developers, and perhaps more relevant to the scientific community, to offer a working environment that enables creative new computational chemistry possibilities for research, from high-throughput analyses to difficult and complex individual computational research problems. We will continue to use our prior experience with GAMESS, QMView, web portals, as well as new workflow and grid middleware technologies as our guide in the design. Careful choice of protocols is crucial to ensure the reusability of developed infrastructure across an entire spectrum of computational science applications. The main parts of our software are planned to be available free of charge under an open source license.

ACKNOWLEDGMENTS

K.K.B. and J.P.G. acknowledge NSF-ANI-0223043, NSF-PRAGMA, and NIH-NBCR-RR08605 for support. W.S. thanks for support by J. Andrew McCammon and by a fellowship within the Postdoc Program of the German Academic Exchange Service (DAAD) during her time at the University of California, San Diego (UCSD). We are grateful to Laura Brovold, UCSD, for some of the benchmark studies, to Chris Kondric, UCSD, for his protein–ligand docking work, while funded by the PRAGMA-Prime program, and to Yang Zhao, University of California, Berkeley, for her help in the computational chemistry workflow project, under project funding NSF-DBI-0078296. Nimrod/G has been funded partly by the Cooperative Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science, and Training). Nimrod/O has been funded by the Australian Research Council. The UZ-ROCKclimber group, the SDSC-ROCKS cluster group, the PRAGMA collaboration,

NBCR, the W.M. Keck Foundation, and UCSD all provided computing resources for the work discussed here.

REFERENCES

1. M. Schmidt, K. K. Baldridge, J. A. Boatz, S. Elbert, M. Gordon, J. H. Jenson, S. Koeski, N. Matsunaga, K. A. Nguyen, S. J. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery, The general atomic and molecular electronic structure system, *J. Comp. Chem.*, 14, 1347 (1993).
2. *The General Atomic and Molecular Electronic Structure System* (*GAMESS*), Ames Laboratory, Iowa State University, Ames, Iowa, 2004. Available at http://www.msg.ameslab.gov/GAMESS/GAMESS.html.
3. G. D. Fletcher, M. W. Schmidt, B. M. Bode, and M. S. Gordon, The distributed data interface in GAMESS, *Comp. Phys. Commun.*, 128, 190 (2000).
4. *MPICH — A Portable Implementation of MPI,* Argonne National Laboratory, Argonne, Illinois, 2004. Available at http://www-unix.mcs.anl.gov/mpi/mpich/.
5. *OpenPBS*, Altair Engineering, Inc., Troy, Michigan, 2004. Available at http://www.openpbs.org/.
6. *Sun Grid Engine*, Sun Microsystems, Inc., Santa Clara, California, 2004. Available at http://gridengine.sunsource.net/.
7. P. M. Papadopoulos, C. A. Papadopoulos, M. J. Katz, W. J. Link, and G. Bruno, Configuring Large High-Performance Clusters at Lightspeed: A Case Study, *International Journal of High Performance Computing Applications*, 18, 317 (2004).
8. K. K. Baldridge and J. P. Greenberg, QMView: a computational 3D visualization tool at the interface between molecules and man, *J. Mol. Graphics*, 13, 63 (1995).
9. M. Thomas, S. Mock, M. Dahan, K. Mueller, D. Sutton, and J. R. Boisseau, The Gridport Toolkit: A System for Building Grid Portals, *10th IEEE International Symposium on High Performance Computing*, San Francisco, CA, 2001.
10. K. K. Baldridge and J. P. Greenberg, *GAMESS Web Portal*, 2004. Available at http://gridport.npaci.edu/gamess/.
11. A. K. Rajasekar and M. Wan, SRB and SRBRack — Components of a Virtual Data Grid Architecture, *Advanced Simulation Technologies Conference (ASTC02)*, San Diego, California, 2002.
12. I. Foster and C. Kesselman, Globus: a metacomputing infrastructure toolkit. *Int. J. Supercomput. Appl.*, 11, 115 (1997).
13. J. P. Greenberg, GAMESS/QMVIEW, *Computational Representation of Bio-Molecules (CRBM) Workshop*, University of California, San Diego, 2003.
14. K. Baldridge and J. Greenberg, Representation of Computational Quantum Chemistry Data in a Structured Format and Incorporation into a Scientific Workflow, *Toward a Common Data and Command Representation for Quantum Chemistry*, Edinburgh, UK, 2004. Available at http://www.nesc.ac.uk/talks/394/data_rep_apr4.pdf.
15. J. Greenberg, S. Mock, M. J. Katz, G. Bruno, F. D. Sacerdoti, P. Papadopoulos, and K. Baldridge, Incorporation of Middleware and Grid Technologies to Enhance Usability in

Computational Chemistry Applications, *Computational Science — ICCS 2004: Proceedings of 4th International Conference, Part I,* Krakow, Poland, Springer-Verlag, Heidelberg, 2004.

16. I. Foster and C. Kesselman, Eds., *The Grid 2: Blueprint for a New Computing Infrastructure*, 2nd ed., Morgan Kaufmann, San Francisco, 2003.
17. F. Berman, G. Fox, and A. J. G. Hey, Eds., *Grid Computing: Making the Global Infrastructure a Reality*, Wiley, 2003.
18. *TeraGrid*, 2004. Availale at http://www.teragrid.org/.
19. *Swiss Bio Grid*, 2004. Available at http://www.swissbiogrid.org/.
20. *Nimrod: Tools for Distributed Parametric Modelling*, Monash University, Clayton, Australia, 2004. Available at http://www.csse.monash.edu.au/∼davida/nimrod/.
21. K. K. Baldridge, J. P. Greenberg, W. Sudholt, S. Mock, K. Bhatia, A. Birnbaum, C. Amoreira, Y. Potier, and M. Taufer, The Computational Chemistry Prototyping Environment, *Proceedings of the IEEE*, 93, 510 (2005).
22. D. Abramson, R. Sosic, J. Giddy, and B. Hall, Nimrod: A Tool for Performing Parametised Simulations Using Distributed Workstations, *The 4th IEEE Symposium on High Performance Distributed Computing*, Virginia, 1995.
23. D. Abramson, J. Giddy, and L. Kotler, High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? *International Parallel and Distributed Processing Symposium (IPDPS)*, Cancun, Mexico, 2000.
24. D. Abramson, A. Lewis, and T. Peachy, Nimrod/O: A Tool for Automatic Design Optimization, *The 4th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2000)*, Hong Kong, 2000.
25. W. Sudholt, K. K. Baldridge, D. Abramson, C. Enticott, and S. Garic, Parameter scan of an effective group difference pseudopotential using grid computing, *New Generation Comput.*, 22, 137 (2004).
26. W. Sudholt, K. K. Baldridge, D. Abramson, C. Enticott, and S. Garic, Applying Grid Computing to the Parameter Sweep of a Group Difference Pseudopotential, *Computational Science — ICCS 2004: Proceeding of 4th International Conference, Part I*, Krakow, Poland, Springer-Verlag, Heidelberg, 2004.
27. *Pacific Rim Applications and Grid Middleware Assembly (PRAGMA)*, 2004. Available at http://www.pragma-grid.net/.
28. *The RCSB Protein Data Bank*, 2004. Available at http://www.rcsb.org/pdb/.
29. N. A. Baker, D. Sept, S. Joseph, M. J. Holst, and J. A. McCammon, Electrostatics of nanosystems: application to microtubules and the ribosome, *Proc. Natl. Acad. Sci.*, 98, 10037 (2001).
30. *APBS: Adaptive Poisson-Boltzmann Solver*, Washington University of St. Louis, St. Louis, Missouri, 2004. Available at http://agave.wustl.edu/apbs/.
31. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock, Kepler: Towards a Grid-Enabled System for Scientific Workflows, *Workflow in Grid Systems Workshop in GGF10 — The Tenth Global Grid Forum*, Berlin, Germany, 2004.
32. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock, Kepler: an Extensible System for Design and Execution of Scientific Workflows, *16th International Conference on Scientific and Statistical Database Management (SSDBM)*, Santorini Island, Greece, 2004.

33. *Kepler: A System for Scientific Workflows*, 2004. Available at http://kepler.ecoinformatics.org/ and http://www.kepler-project.org/.
34. *Ptolemy II: Heterogenous Modeling and Design*, University of California, Berkeley, California, 2004. Availale at http://ptolemy.eecs.berkeley.edu/ptolemyII/.
35. *Open Babel: A Package to Decypher Computational Chemistry*, 2004. Available at http://openbabel.sourceforge.net/.
36. *Extensible Markup Language (XML)*, W3C, 2004. Available at http://www.w3.org/XML/.
37. P. Murray-Rust, H. S. Rzepa, and M. Wright, Development of chemical markup language (CML) as a system for handling complex chemical content, *New J. Chem.*, 25, 618 (2001).
38. *Chemical Markup Language (CML)*, 2004. Available at http://www.xml-cml.org/ and http://cml.sourceforge.net/.
39. *Cambridge Structural Database (CSD)*, Cambridge Crystallographic Data Centre (CCDC): Cambridge, UK, 2004. Available at http://www.ccdc.cam.ac.uk/products/csd/.
40. *DB2 Product Family*, IBM, 2004. Available at http://www-306.ibm.com/software/data/db2/.
41. *Resurgence Project Home Page*, University of Zurich, Zurich, Switzerland, 2004. Available at http://www.Baldridge.unizh.ch/resurgence.
42. *INFORMNET: INformation Flow and Operation Resource Management on the NET*, San Diego Supercomputer Center, San Diego, California, 2004. Available at http://chemport.sdsc.edu/informnet/.

The Virtual Laboratory: A Toolset to Enable Distributed Molecular Modelling for Drug Design on the World-Wide Grid

Rajkumar Buyya[1], Kim Branson[2], Jon Giddy[3], and David Abramson[4]

[1]Dept. of Computer Science and Software Engineering
The University of Melbourne
Parkville, Melbourne, Australia
raj@cs.mu.oz.au

[2] Structural Biology
Walter and Eliza Hall Institute
Royal Parade, Parkville, Melbourne
kbranson@wehi.edu.au

[3]Welsh e-Science Centre
Department of Computer Science, Cardiff
University, Cardiff, UK
J.P.Giddy@cs.cf.ac.uk

[4]School of Computer Science and Software Eng.
Monash University, Caulfield Campus
Melbourne, Australia
davida@csse.monash.edu.au

Abstract: Computational Grids are emerging as a new paradigm for sharing and aggregation of geographically distributed resources for solving large-scale compute and data intensive problems in science, engineering, and commerce. However, application development, resource management and scheduling in these environments is a complex undertaking. In this paper, we illustrate the development of a virtual laboratory environment by leveraging existing Grid technologies to enable molecular modelling for drug design on geographically distributed resources. It involves screening millions of compounds in the chemical database (CDB) against a protein target to identify those with potential use for drug design. We have used the Nimrod-G parameter specification language to transform the existing molecular docking application into a parameter sweep application for executing on distributed systems. We have developed new tools for enabling access to ligand records/molecules in the CDB from remote resources. The Nimrod-G resource broker along with molecule CDB data broker is used for scheduling and on-demand processing of docking jobs on the World-Wide Grid (WWG) resources. The results demonstrate the ease of use and power of the Nimrod-G and virtual laboratory tools for grid computing.

1 Introduction

Computational Grids [1] enable the sharing of a wide variety of geographically distributed resources including supercomputers, storage systems, databases, data sources, and specialized devices owned by different organizations in order to create virtual enterprises and organizations. They allow *selection* and *aggregation* of distributed resources across multiple organizations for solving large-scale computational and data intensive problems in science, engineering, and commerce. The parallel processing of applications on wide-area distributed systems provide a scalable computing power. This enables exploration of large problems with huge data sets, which is essential for creating new insights into the problem. Molecular modelling for drug design is one of the scientific applications that can benefit from the availability of a large computational capability.

Drug discovery is an extended process that can take as many as 15 years from the first compound synthesis in the laboratory until the therapeutic agent, or drug, is brought to market [11]. Reducing the research timeline in the discovery stage is a key priority for pharmaceutical companies worldwide. Many such companies are trying to achieve this goal through the application and integration of advanced technologies such as computational biology, chemistry, computer graphics, and high performance computing (HPC). Molecular modelling has emerged as a popular methodology for drug design—it can combine computational chemistry and computer graphics. Molecular modelling can be implemented as a

master-worker parallel application, which can take advantage of HPC technologies such as clusters [2] and Grids for large-scale data exploration.

Drug design using molecular modelling techniques involve screening a very large number (of the order of a million) of ligand[1] records or molecules of compounds in a chemical database (CDB) to identify those that are potential drugs. This process is called molecular *docking*. It helps scientists in predicting how small molecules, such as substrates or drug candidates, bind to an enzyme or a protein receptor of known 3D structure (see Figure 1). Docking each molecule in the target chemical database is both a compute and data intensive task. It is our goal to use Grid technologies to provide cheap and efficient solutions for the execution of molecular docking tasks on large-scale, wide-area parallel and distributed systems.

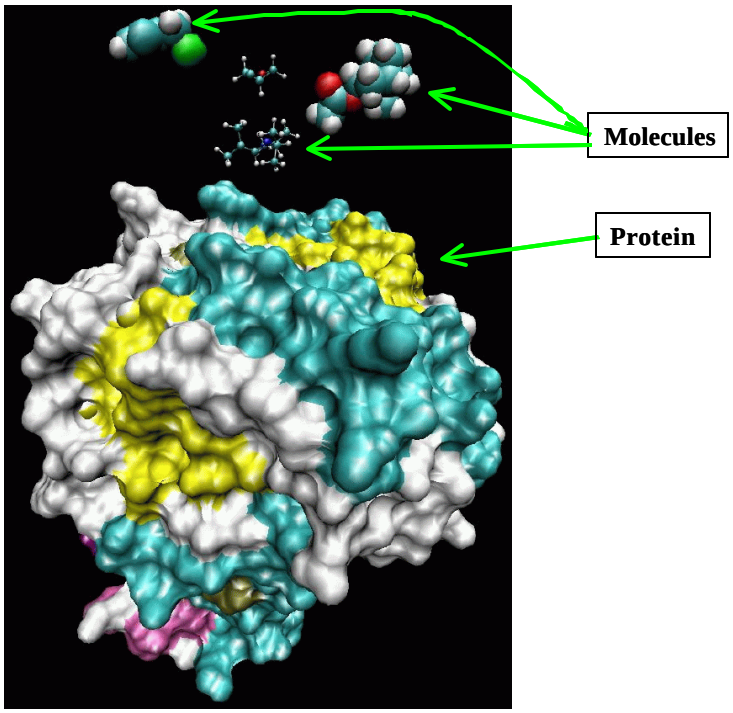


Figure 1: X-ray crystal structure of a target protein receptor and small molecules to be docked.

While performing docking, information about the molecule must be extracted from one of the many large chemical databases. As each chemical database requires storage space in the order of hundreds of megabytes to terabytes, it is not feasible to transfer the chemical database to all resources in the Grid. Also, each docking job only needs a ligand or module record, not the whole database. Therefore, access to a chemical database must be provided as a *network service* (see Figure 2). The chemical databases need to be selectively replicated on a few nodes to avoid any bottleneck due to providing access to the database from a single source. Intelligent mechanisms (e.g., CDB broker) need to be supported for selecting optimal sources for CDB services depending on the location of resources selected for processing docking jobs.

Fundamentally, drug design is a *computational and data challenge* problem since it involves screening millions of compounds in chemical databases. Screening each compound, depending on structural complexity, can take from a few minutes to hours on a standard PC, which means screening all compounds in a single database can take years! For example, we are looking into a drug design problem that involves screening 180,000 compounds. Each job screening a compound is expected to take up to 3 hours of execution time on a desktop computer (e.g., Pentium-based Linux/Windows PC). That means, if we aim to screen all these compounds on a single PC, it can take up to 540000 hours, which is roughly equivalent to 61 years! If we use a typical cluster-based supercomputer with 64 nodes, we can solve this problem in one year. The problem can be solved with a large scale Grid of hundreds of supercomputers within a day. If we use a massive network of peer-to-peer style Grid computing infrastructure such as SETI@Home [19], the

[1] An ion, a molecule, or a molecular group that binds to another chemical entity to form a larger complex.

drug discovery problem could be solved within a few hours.

The rest of this paper is organized as follows. A high-level operational model for molecular modelling on the Grid is presented in Section 2. A layered architecture for building the Virtual Laboratory environment is presented in Section 3. It leverages the existing Grid technologies and supports new tools that are essential for Grid-enabling the chemical database and the docking application on distributed resources. Formulation of molecular docking as a parameter sweep application is presented in Section 4. The results of two experiments on scheduling molecular docking jobs for processing on the WWG (World Wide Grid) [18] testbed resources are presented in Section 5. The final section summarizes the paper along with suggestions for future works.

2 Operational Model

The Virtual Laboratory tools transform the existing molecular modelling application (without the need for making any changes to it) into a parameter sweep application for executing jobs docking molecules in the CDBs in parallel on distributed resources. The parameterized application contains multiple independent jobs, each screening different compounds to identify their drug potential. These jobs are computationally intensive in nature and only a small proportion of the execution time is spent on data communication (e.g., fetching molecular information on demand from remote databases). Applications expressed with this task-farming computational model have high *computation to communication* ratio. Hence, they can tolerate high network latency, which makes them suitable for executing in parallel on Internet-wide distributed resources.

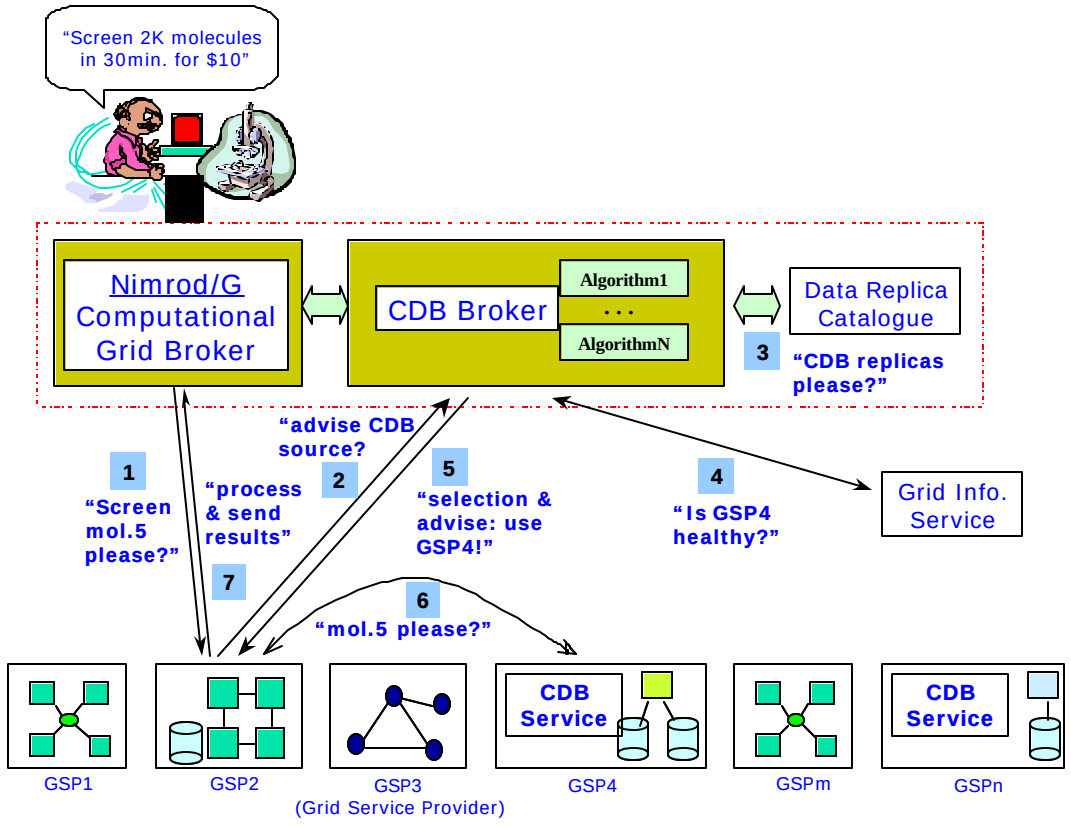


Figure 2: Resource brokering architecture for screening molecules on distributed resources.

A high-level operation model of docking molecules on the Grid is shown in Figure 2. The drug designer formulates the molecular docking problem, submits the application to the Grid resource broker (e.g., Nimrod-G [4][5]) along with performance and optimisation requirements—"screen 2000 molecules within 30 minutes and the available budget for processing is $10" [5]. The broker discovers resources, establishes their cost and capability, and then prepares a schedule to map docking jobs to resources. Let us say, it

identified a GSP (Grid Service Provider), say GSP2, and assigned a job of screening a molecule 5 to it. A job has a task specification that specifies a list of operations to be performed. To process a job on GSP2, the broker dispatcher deploys its Agent on resource GSP2. The agent executes a list of commands specified in the job's task specification. A typical task specification contains necessary commands to copy executables and input files from the user machine, substitution of parameters declared in the input file, execution of the program, and finally copying results back to the user. It can also contain special commands for accessing the input data from the remote database. For example, a docking task can contain a special command (e.g., an instruction to fetch molecule record from the CDB) to make a request to the data broker (e.g., CDB broker) for a molecule record. The data broker looks at the replica catalogue for a list of sites providing CDB services, checks the status of those sites, and selects a suitable site (e.g., a node with fast network connectivity) and recommends the same. The molecule fetch command can then request the CDB service provider for a molecule record and write the molecule structure to a file that acts as an input to the docking program. After executing the docking program, the agent executes commands related to copying docking results to the user home node.

3 Architecture – The Software Stack

The Virtual Laboratory builds on the existing Grid technologies and tools for performing data intensive computing on distributed resources. It provides new tools for managing and accessing remote chemical databases as a network service. There are many scientific and commercial applications (e.g., molecular modelling, high-energy physics events processing, and financial investment risk-analysis) that explore a range of scenarios. Instead of explicitly developing them as parallel applications using interfaces such as MPI, they can be composed as parameter sweep applications using tools such as Nimrod [3]. Such application jobs can be executed in parallel on distributed resources using the Nimrod-G resource broker (see Figure 2). A layered architecture and the software stack essential for performing molecular modelling on distributed resources is depicted in Figure 3. The components of the Virtual Laboratory software stack are:

- The DOCK software for Molecular Modelling [15].
- The Nimrod Parameter Modelling Tools [3][17] for enabling DOCK as a parameter sweep application.
- The Nimrod-G Grid Resource Broker [4][5] for scheduling DOCK jobs on the Grid.
- Chemical Database (CDB) Management and Intelligent Access Tools:
    - CDB database lookup/Index table generation.
    - CDB and associated index-table replication.
    - CDB replica catalogue for CDB resource discovery.
    - CDB servers for providing CDB services
    - CDB broker for selecting a suitable CDB service (Replica Selection).
    - CDB clients for fetching molecular records (Data Movement).
- The GrACE software for resource trading toolkit [6].
- The Globus middleware for secure and uniform access to distributed resources [9].

The Grid resources (e.g., multiprocessors or clusters) at each location are generally presented as a single entity using resource management systems such as OS-fork, LSF, Condor, and SGE.

3.1 Docking Code

The original docking code developed by researchers at the University of California in San Francisco (UCSF) is one of the most popular molecular docking applications [10]. The program evaluates the chemical and geometric complementarities between a small molecule and a macromolecular binding site. It explores ways in which two molecules, such as a drug and an enzyme or protein receptor, might fit together. Compounds that might bind tightly to the target receptor must have complementary chemical and spatial natures. Thus docking can be seen as a 3 dimensional puzzle searching for pieces that will fit into the receptor site. It is important to be able to identify small molecules (compounds), which may bind to a target macromolecule. This is because a compound, which binds to a biological macromolecule, may modulate its function, and with further development eventually become a drug candidate. An example of such a drug is the anti influenza drug Relenza which functions by binding to influenza virus attachment proteins thus preventing viral infection.

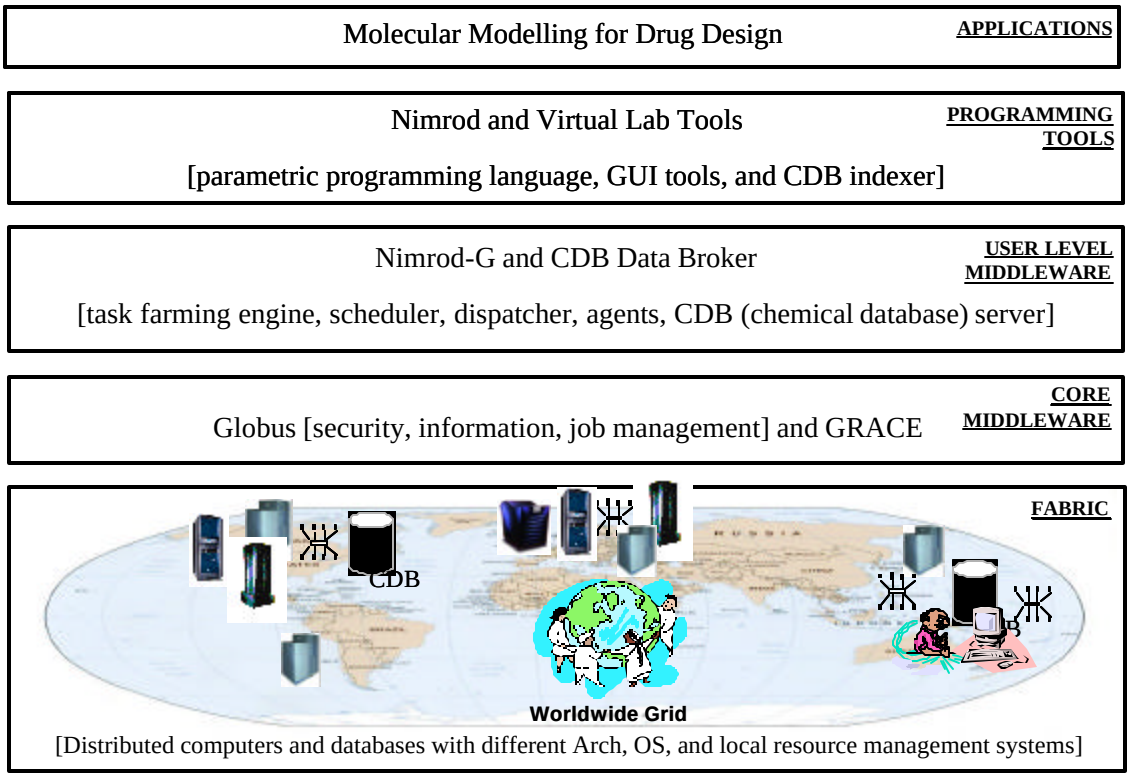


Figure 3: Layered architecture of Virtual Laboratory for drug design.

The relationship between the key programs in the dock suite is depicted in Figure 4 (source [10]). The receptor coordinates at the top represent the three-dimensional (3D) structure of a protein. The molecular modeller identifies the active site, and other sites of interest, and uses the program *sphgen* to generate the sphere centers, which fill the site [14]. The program *grid* generates the scoring grids [15]. The program *dock* matches spheres (generated by sphgen) with ligand atoms and uses scoring grids (from grid) to evaluate ligand orientations [14] [15]. It also minimizes energy-based scores [16]. The focus of our work is on docking molecules in CDB with receptors to identify potential compounds that act as a *drug*. Hence, discussion in this paper is centered on the execution of the program *dock* as a parameter sweep application on world-wide distributed resources.

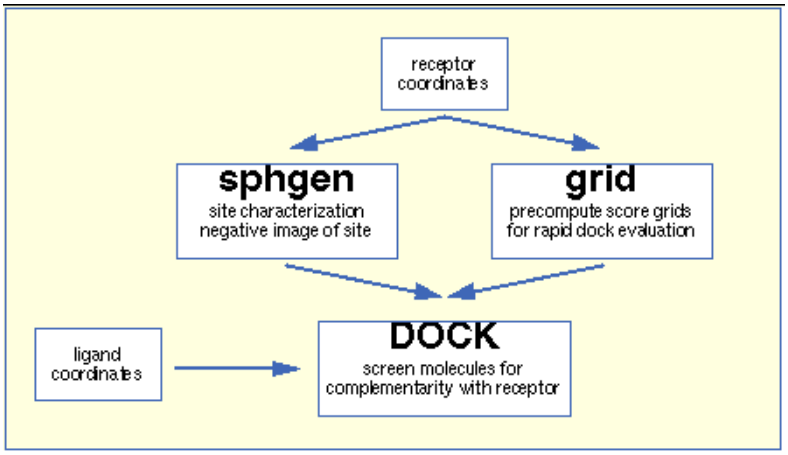


Figure 4: Relation between key programs in the *dock* suite.

The docking code is highly portable—we have been able to compile and produce executables for Sun-

Solaris, PC Linux, SGI IRIX, and Compaq Alpha/OSF1 architectures. For docking on heterogeneous resources, the Nimrod-G broker selects the correct executable automatically based on the resources it discovers at runtime.

3.2 Nimrod-G Tools

The Nimrod-G toolkit provides a suite of tools and services for creating parameter sweep applications, managing resources and scheduling applications on the world-wide distributed resources. It supports a simple declarative programming language and GUI tools for creating scripts and parameterization of application input data files, and a programmable grid resource broker for processing jobs on grid resources.

Tools for Creating Parameter Sweep Applications

The Nimrod-G toolkit provides a suite of tools and services for creating parameter sweep applications, managing resources and scheduling applications on the world-wide distributed resources. It provides a simple *declarative programming language* or GUI tools for parameterization of application input data files and creation of task-script to be performed by each job; and a programmable *grid resource broker* for processing jobs on Grid resources.

The steps involved in distributed parametric execution are:

a) parameterise input files,

b) prepare a plan file containing the commands that define parameters and their values,

c) generate a run file, which converts the generic plan file to a detailed list of jobs,

d) schedule jobs for processing on distributed machines, and

e) control and monitor the execution of the jobs.

The application execution environment handles online creation of input files and command line arguments through parameter substitution. The GUI tools supported by enFuzion, a commercial version of Nimrod, can also be used for parameterising applications. enFuzion uses the same syntax as Nimrod for parameter specification [17]. Both Nimrod and enFuzion have been successfully used for performing parameter studies in a single administrative domain such as clusters. Nimrod-G [4][5] extends the capabilities of Nimrod and EnFuzion with the addition of powerful resource discovery, trading, and scheduling algorithms [7]. In Section 4, we discuss the capabilities of Nimrod-G tools by composing a molecular modelling program as a parameter sweep application for docking compounds in CDB databases and processing docking jobs on the Grid.

Nimrod-G Grid Resource Broker for scheduling DOCK jobs on Grid

The Nimrod-G Resource broker identifies the user and application processing requirements and selects Grid resource combination in such a way that the user requirements are met [4][5][26]. It performs resource discovery, selection, scheduling, dispatching of docking jobs to remote resource, starting and managing the execution of jobs, and gathering results back to the home node. The sub-modules of our resource broker are,

- A persistent task farming engine,
- A grid explorer for resource discovery,
- A resource trading manager for establishing access price,
- A scheduler that maps jobs to resources using scheduling algorithms.
- A dispatcher and actuators for deploying agents on grid resources; and
- Agents for managing execution of Nimrod-G jobs on grid resources.

When Nimrod-G deploys its Agents on the Grid node, it is submitted to the local resource manager, which then allocates a compute node[2] to it for executing the job.

The Nimrod-G components, grid explorer for resource discovery and dispatcher for submitting jobs to remote resource, are implemented using the Globus GIS and GRAM services. The dispatcher actually initiates the execution of Nimrod-G agents on remote resources, which takes care of all the operations associated with the execution of an assigned job. The interaction between the components of the Nimrod-G runtime machinery and Grid services during runtime is shown in Figure 5. The Nimrod-G broker supports

[2] It is a node/CPU that the local resource manager allocates to the Nimrod-G agent for job processing.

deadline and budget constrained (DBC) scheduling algorithms driven by a computational economy and user requirements [7]. In Section 5, we discuss the results of the Nimrod-G broker scheduling a molecular modelling application on the Grid with DBC time and cost optimization scheduling algorithms.

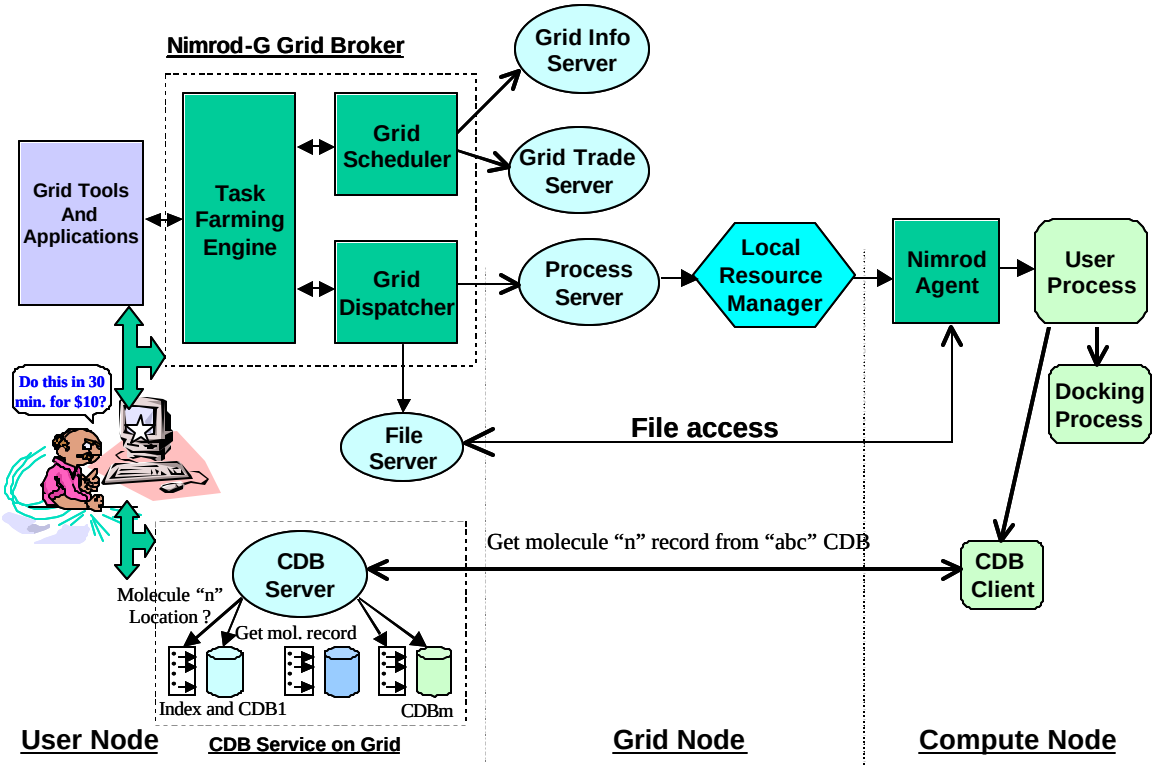


Figure 5: Deployment of Virtual Laboratory components at runtime and their interaction.

3.3 Chemical Database Management and Intelligent Access Tools

The chemical databases contain records of a large number of molecules from commercially available organic synthesis libraries, and natural product databases. The molecules in the CDB are represented in MOL2 file (.mol2) format [12], which is a portable representation of a SYBYL [13] molecule. The MOL2 file is an ASCII file that contains all the information needed to reconstruct a SYBYL molecule. Each ligand record in a chemical database represents the three-dimensional structural information of a compound. The numbers of compounds in each CDB can be in the order of tens of thousands and the database size be anywhere from tens of Megabytes to Gigabytes and even Terabytes. We have developed tools for turning the CDB into a network service and accessing them from remote resources. They include tools for indexing ligand records in the CDB, a multithreaded CDB Server for serving requests for molecule records, and a tool for fetching molecule / ligand record from remote CDB via the network [20].

When a chemical database is available from more than one source (replica site), a suitable strategy such as a source with high network speed or lightly loaded, can be used for selecting one of them. It is likely that multiple users from different locations issue requests for accessing the CDB, the server should be able to process such simultaneous requests concurrently. Therefore, we have developed a multithreaded CDB server that can service requests from multiple users concurrently. An interaction between a Grid node and a node running the CDB server while performing docking is shown in Figure 5. We developed and implemented protocols shown in Figure 6 for interaction between interaction between the CDB clients and the server. Both figures illustrate the operational model and the flow of control between CDB clients and servers.

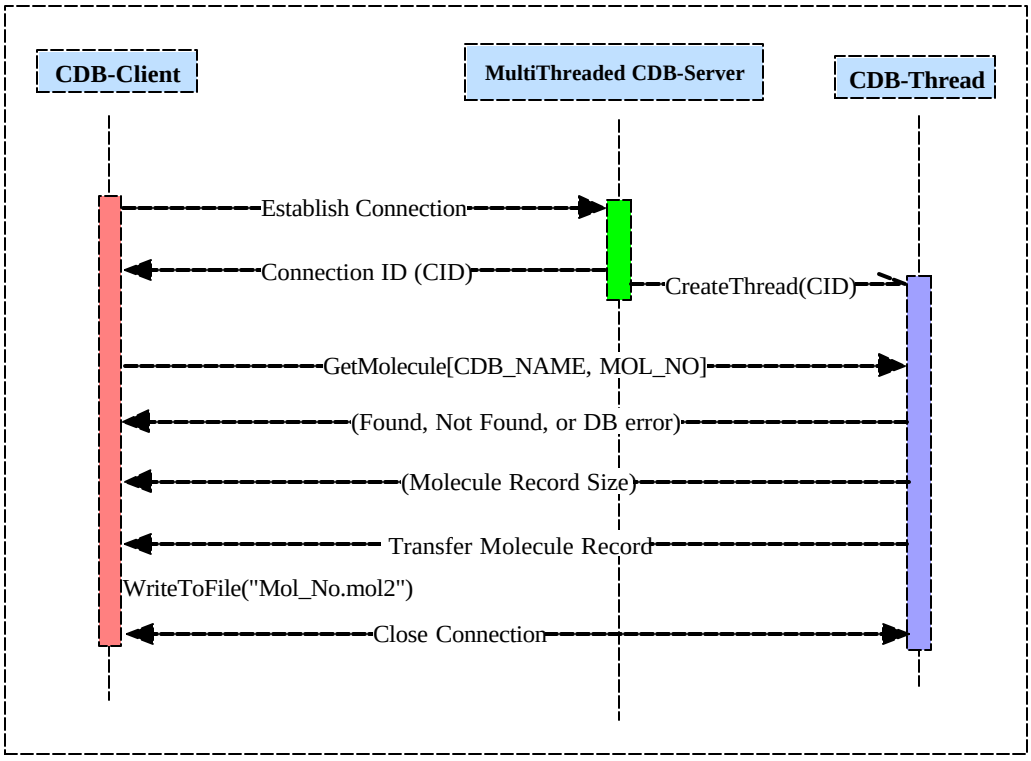


Figure 6: Protocols for Interaction between the CDB clients and the server.

When the Nimrod-G schedules a docking job for processing on one of the Grid resources, it actually submits an agent to Globus GRAM, which acts as a process server. The process server either executes it by forking it as process on time shared system or submits to the site resource manager such as PBS, LSF, and Condor, which allocates a compute node for the agent and starts its execution. The agent contacts the Nimrod-G dispatcher for job task information, which contains instructions for executing a job. It copies input files, performs parameter substitution, executes programs (e.g., CDB client to fetch a molecule record from the remote CDB server and docking program), and ships results back to the Nimrod-G user. When the CDB server receives a request for molecule record, it reads the molecule record from the chemical database and sends back to the client.

Instead of searching molecule records sequentially in a database, we have built tools for creating index-tables for each CDB represent using the MOL2 format along with the record size information. The CDB index file, organized in in binary format, contains the starting address (byte location) of a molecule record and record size of all molecules in sequence. When a molecule record is requested, the CDB server first looks at the CDB index file to identify the record location and its size. It then fetches the molecule record from the CDB file with a single read operation and thus improving the access and response speed.

It is possible to screen virtual combinatorial databases in their entirety. This methodology allows only the potential compounds to be subjected to physical (manual) screening and/or synthesis in laboratories, which is extremely time-consuming and resource-intensive.

4 Application Composition

A docking application having an ability to screen a molecule for each execution can be comp osed as a task-farming, parameter sweep application for distributed execution. This can be achieved by using Nimrod-G parameter specification language to parameterize docking application input data and files. There is no need to make any changes to the existing (sequential) docking application nor it needs to be developed as parallel application explicitly for distributed execution. The users just need to parameterize the input data and files appropriately and define a Nimrod-G plan file once. Note the valu e of parameters can be changed while launching the application execution. The plan file specifies the parametric tasks and the types of the parameters and their values for these tasks. A parametric task consists of a script defined using a sequence

of simple commands, providing the ability to copy files to and from the remote node, perform parameter substitutions in input files, execute certain programs, and copy output files back to the user home node. The parametric plan can be submitted to the Nimrod-G runtime machinery, which creates independent docking jobs, and schedules jobs for concurrent execution on distributed resources. It takes care of replacing the actual value of parameters in the parameterized input files before executing docking jobs.

A sample configuration input file of the docking application is shown in Figure 7. It specifies docking configuration parameters and molecule to be docked by indicating a name of the file in which molecule record is stored using the parameter variable "ligand_atom_file". To perform a parameter sweep of different molecules, the value specified by the parameter variable "ligand_atom_file" needs to be parameterized. This is accomplished by replacing the current value, which represents the name of a file containing molecule record, by a substitution place marker. The place marker T consists of a dollar-sign ($) followed by the name of the parameter controlling the substitution, optionally surrounded by braces.

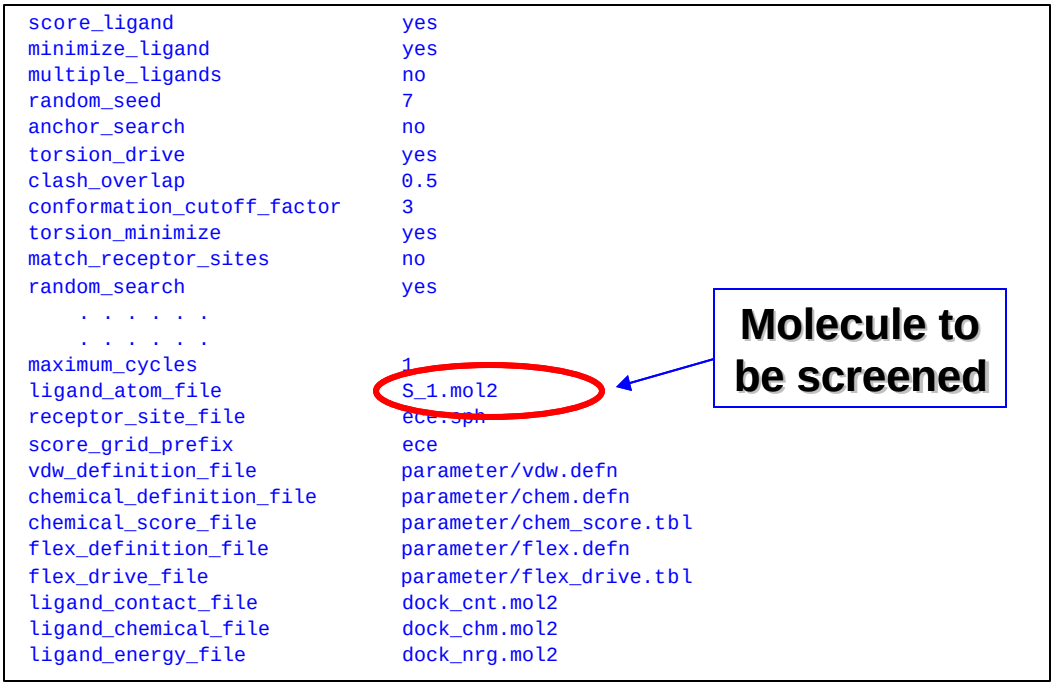


Figure 7: A configuration input file for docking application.

A parameterized input file with several attributes replaced by substitution place markers is shown in Figure 8. For example, a place marker called for the parameter “ligand_number” has replaced the first part of the “ligand_atom_file” attribute value. The actual value of these parameters is defined in the Nimrod-G plan file that contains parameter definition and task specification. The parameter definition section of the plan file is shown in Figure 9. Each parameter is defined by a keyword "parameter", followed by the parameter name, an optional label, and a parameter type. The remaining information on each line defines valid values for the parameter.

```
score_ligand                $score_ligand
minimize_ligand             $minimize_ligand
multiple_ligands            $multiple_ligands
random_seed                 $random_seed
anchor_search               $anchor_search
torsion_drive               $torsion_drive
clash_overlap               $clash_overlap
conformation_cutoff_factor  $conformation_cutoff_factor
torsion_minimize            $torsion_minimize
match_receptor_sites        $match_receptor_sites
random_search               $random_search
    . . . . . .
    . . . . . .
maximum_cycles              $maximum_cycles
ligand_atom_file            ${ligand_number}.mol2
receptor_site_file          $HOME/dock_inputs/${receptor_site_file}
score_grid_prefix           $HOME/dock_inputs/${score_grid_prefix}
vdw_definition_file         vdw.defn
chemical_definition_file    chem.defn
chemical_score_file         chem_score.tbl
flex_definition_file        flex.defn
flex_drive_file             flex_drive.tbl
ligand_contact_file         dock_cnt.mol2
ligand_chemical_file        dock_chm.mol2
ligand_energy_file          dock_nrg.mol2
```


Figure 8: Parameterisation of a configuration input file.

The parameter, for example, "database_name" has a label, and is of type *text*. Its valid values are listed, and the user will be able to select one of the values for the duration of the entire experiment. Most of the remaining parameters are single values, either text strings or integers, selected by the user, but with default values provided if the user does not wish to choose a value.

The range parameter, "ligand_number", used to select the molecule, is defined as an integer variable with bounds. For example, to process the first 2000 molecules in the CDB, this range parameter can vary from 1 to 2000 with the step size of 1.

```
parameter database_name label "database_name" text select oneof "aldrich"
    "maybridge" "maybridge_300" "asinex_egc" "asinex_epc" "asinex_pre"
    "available_chemicals_directory" "inter_bioscreen_s" "inter_bioscreen_n"
    "inter_bioscreen_n_300" "inter_bioscreen_n_500" "biomolecular_research_institute"
    "molecular_science" "molecular_diversity_preservation"
    "national_cancer_institute" "IGF_HITS" "aldrich_300" "molecular_science_500"
    "APP" "ECE" default "aldrich_300";
parameter CDB_SERVER text default "bezek.dstc.monash.edu.au";
parameter CDB_PORT_NO text default "5001";
parameter score_ligand text default "yes";
parameter minimize_ligand text default "yes";
parameter multiple_ligands text default "no";
parameter random_seed integer default 7;
parameter anchor_search text default "no";
parameter torsion_drive text default "yes";
parameter clash_overlap float default 0.5;
parameter conformation_cutoff_factor integer default 5;
parameter torsion_minimize text default "yes";
parameter match_receptor_sites text default "no";
    . . . . . .
    . . . . . .
parameter maximum_cycles integer default 1;
parameter receptor_site_file text default "ece.sph";
parameter score_grid_prefix text default "ece";
parameter ligand_number integer range from 1 to 2000 step 1;
```


Figure 9: A plan file defining parameters type and their values.

```
task nodestart
      copy ./parameter/vdw.defn node:.
      copy ./parameter/chem.defn node:.
      copy ./parameter/chem_score.tbl node:.
      copy ./parameter/flex.defn node:.
      copy ./parameter/flex_drive.tbl node:.
      copy ./dock_inputs/get_molecule node:.
      copy ./dock_inputs/dock_base node:.
endtask
task main
           node:substitute dock_base dock_run
           node:substitute get_molecule get_molecule_fetch
           node:execute sh ./get_molecule_fetch
           node:execute $HOME/bin/dock.$OS -i dock_run -o dock_out
           copy node:dock_out ./results/dock_out.$jobname
           copy node:dock_cnt.mol2 ./results/dock_cnt.mol2.$jobname
           copy node:dock_chm.mol2 ./results/dock_chm.mol2.$jobname
           copy node:dock_nrg.mol2 ./results/dock_nrg.mol2.$jobname
endtask
```

Figure 10: Task definition of docking jobs.

The parameters “receptor_site_file” and “score_grid_prefix” indicate the data input files. Their values indicate that data input files are located in the user home directory on Grid nodes. Instead of pre-staging, these files can be copied at runtime by defining necessary “copy” operations in the job’s “nodestart” or “main” task (see Figure 10). However, it is advisable to copy or “pre-stage” large input files in the beginning of application execution instead of copying them during execution of every job. This saves transmission time particularly when those files are going to be used for docking with many databases.

The plan file is submitted to a job generation tool, such as the EnFuzion Generator, in order to create a run file that contains specific instances of jobs to be run, which is then submitted to the Nimrod-G runtime machinery for processing on the Grid. The run file contains a job for each combination of parameters. Hence the number of jobs is the product of the number of values chosen for each parameter. Since most of the parameters except "ligand_number" are single-valued, they have no effect on the number of jobs.

It is also possible to set concrete values for each of the parameters at runtime when job Generator is invoked. For the parameter "ligand_number", the user may choose not to select all values from 1 to 2000, but may select a subset of these values. By default, this generated 2000 jobs, each docking a single molecule.

The second part of Nimrod-G plan file is task specification that defines a series of operations that each job needs to perform to dock a molecule (see Figure 10). The "nodestart" task is performed once for each remote node. Following that, the files copied during that stage are available to each job when it is started. The "main" task controls the actions performed for each job.

The first line of the "main" task performs parameter substitution on the file "dock_base", creating a file "dock_run". This is the action that replaces the substitution place markers in our input file with the actual values for the job.

As each docking operation is performed on a selected molecule in the CDB database, it is not necessary to copy such large databases on all Grid nodes. Hence, not only is the molecule file named in the configuration file, we also go to particular lengths to copy only the data for the molecule being tested. The executable script "get_molecule_fetch" (see Figure 11) is also created using parameter substitution, and runs the "vlab-cdb-get-molecule" executable, which fetches the molecule record from the *CDB molecule server* based on the parameter "ligand_number". The molecule record is saved in a file whose name is the same as integer value of the “ligand_number” parameter and “mol2” as its extension. For instance, if the parameter ligand_number value is 5, then molecule record will be saved in a file “5.mol2”.

```
#!/bin/sh
$HOME/bin/vlab-cdb-get-molecule.$OS $CDB_SERVER $CDB_PORT_NO ${database_name}.db $ligand_number
```

Figure 11: Parameterisation of script for extracting molecule from CDB.

The main code is the "dock" executable. Note that in the “execute” command, there are pseudo-parameters that do not appear in the plan file. These include environment variables, such as "HOME", as well as other useful parameters, such as "OS" indicating the operating system on the node. This allows us to select the correct executable for the node. If the “dock” executable files do not exist on Grid nodes, they need to be copied at runtime as part of the job’s “nodestart” task similar to copying input files.

The dock_run file created in the substitution step previously is now provided as the input configuration file for the docking process. The output files are then copied back to the local host, and renamed with another pseudo-parameter, the unique "jobname" parameter.

5 Scheduling Experimentations

We have performed scheduling experiments from a grid resource in Australia along with four resources available in Japan and one in USA. Table 1 shows the list of resources and their properties, Grid services, access cost or price in terms of Grid dollar (G$) per CPU-second, and the number of jobs processed on resources with deadline-and-budget constrained (DBC) time optimization (TimeOpt) or cost optimization (CostOpt) strategies. The resource price in terms of G$ is assigned arbitrarily at runtime in these experiments, however, they can be set to match the power of resources and job turn around time as valued in supercomputing centers such as the Manchester computing services [24]. The G$ can be equated to real money or tokens charged to users for accessing resources. In the current scenario, the users get allocation of tokens via funding from the project sponsoring agents or partnerships. There are supercomputing centers that sell tokens to commercial users and the value of tokens correspond to the quantity of resource allocations. It is also possible to price resources based on the real world economic models [25] that are driven by the supply and demand for resources.

Table 1: The WWG testbed resources used in scheduling experiments, job execution and costing.

Organization & Location	Vendor, Resource Type, # CPU, OS, hostname	Grid Services and Fabric, Role	Price (G$/CPU sec.)	Number of Jobs Executed	
				TimeOpt	CostOpt
Monash University, Melbourne, Australia	Sun: Ultra-1, 1 node, bezek.dstc.monash.edu.au	Globus, Nimrod-G, CDB Server, Fork (Master node)	--	--	--
AIST, Tokyo, Japan	Sun: Ultra-4, 4 nodes, Solaris, hpc420.hpcc.jp	Globus, GTS, Fork (Worker node)	1	44	102
AIST, Tokyo, Japan	Sun: Ultra-4, 4 nodes, Solaris, hpc420-1.hpcc.jp	Globus, GTS, Fork (Worker node)	2	41	41
AIST, Tokyo, Japan	Sun: Ultra-4, 4 nodes, Solaris, hpc420-2.hpcc.jp	Globus, GTS, Fork (Worker node)	1	42	39
AIST, Tokyo, Japan	Sun: Ultra-2, 2 nodes, Solaris, hpc220-2.hpcc.jp	Globus, GTS, Fork (Worker node)	3	11	4
Argonne National Lab, Chicago, USA	Sun: Ultra -8, 8 nodes, Solaris, pitcairn.mcs.anl.gov	Globus, GTS, Fork (Worker node)	1	62	14
		Total Experiment Cost (G$)		17702	14277
		Time to Finish Experiment (Min.)		34.00	59.30

We have performed a trial screening 200 molecules (from the *aldrich_300* CDB) on a target receptor called endothelin converting enzyme (ECE), which is involved in hypotension. The three dimensional structure of the receptor is derived from homology modelling using related receptor structures whose three

dimensional structures have been solved by X-ray crystallography experiments. In these experimentations, for faster evaluation purpose, the range parameter "ligand_number" is defined with the bounds 1 and 200 and the step size as 1, which produces 200 jobs for docking molecules. As shown in Figure 12, the *dock* program takes two different types of inputs files: a) *common input files*, the same files are required for all docking jobs and b) *ligand specific input files*, which vary from one job to another. The large common input files (receptor structure and pre-calculated score potentials) are pre-staged on resources instead of copying them at runtime. The files are copied using the *globus-rcp* command and stored in the directory location "$HOME/dock_inputs/" on resources as specified by the parameters "receptor_site_file" and "score_grid_prefix" (see Figure 8). The two application-specific executable files, "dock" and "vlab-cdb-get-molecule" invoked in the task scripts (see Figure 10 and Figure 11) are also pre-staged. The executable files are stored in the "$HOME/bin/" directory on resources.

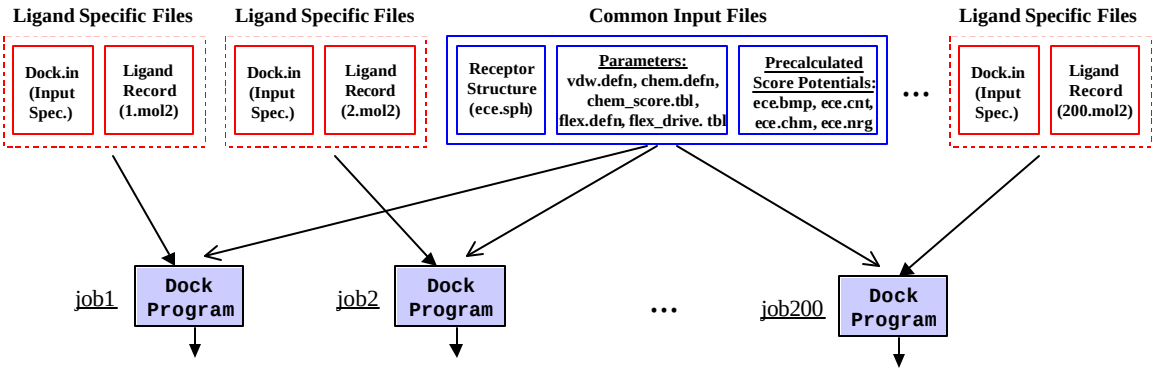


Figure 12: Static and Dynamic Input Files of Docking program.

We conducted deadline and budget constrained scheduling experiments for two different optimization strategies [7]:

1. Optimize for Time - this strategy aims to produce results at the earliest possible time before a deadline, and within a budget limit. It process as many jobs as possible cheapest resources for the deadline period and uses expensive ones just to meet the deadline.
2. Optimize for Cost - this strategy aims to minimize the cost of execution (spending from the given budget) and complete the experiment on or before the deadline. It uses all resources aggressively as long as it can afford them and tries to process all jobs at the earlier possible time.

In both experiments, we have set 60 minutes as the deadline limit and 50,000 G$ as the budget limit at runtime using the Nimrod-G scheduler steering and control monitor. The value of these constraints can be changed at anytime during the execution, of course not less than the time and budget that is already spent!

The first experiment, *Optimize for Time* scheduling, was performed on November 3, 2001 at 23:23:00, Australian Eastern Standard Time (AEST), with a 60-minute deadline and finished on November 3, 2001 by 23:57:00. A snapshot of the Nimrod-G monitoring and steering client taken a few minutes (~5min.) before the completion of application processing is shown in Figure 13. This experiment took 34 minutes to finish the processing of all jobs using resources available at that time with an expense of 17,702 G$. Figure 15 shows the number of jobs processed on different resources selected depending on their cost and availability. Figure 16 shows the corresponding expenses of processing on resources. Figure 17 shows the number of jobs in execution on resources at different times. From the graphs it can be observed that the broker selected resources to ensure that the experiment was completed at the earliest possible time given the current availability of resources and the budget limitations. After 30 minutes, it discovered that it could still complete early without using the most expensive resource, hpc220-2.hpcc.jp.

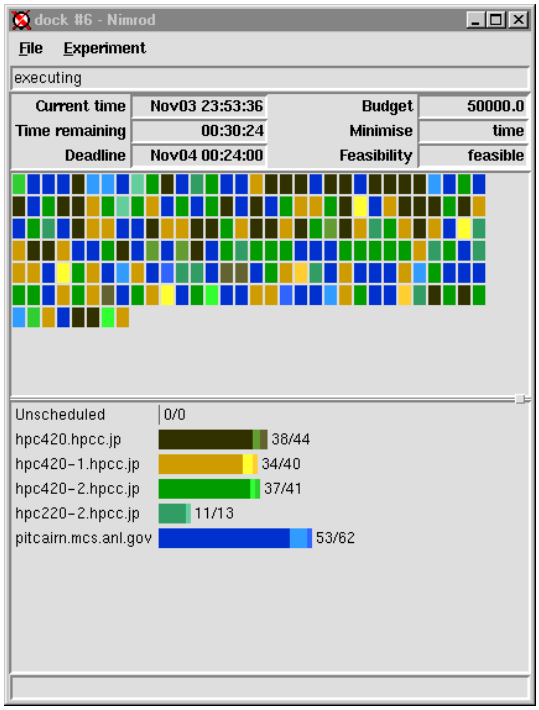


Figure 13: A snapshot of the Nimrod-G monitor during "Optimize for Time" scheduling.

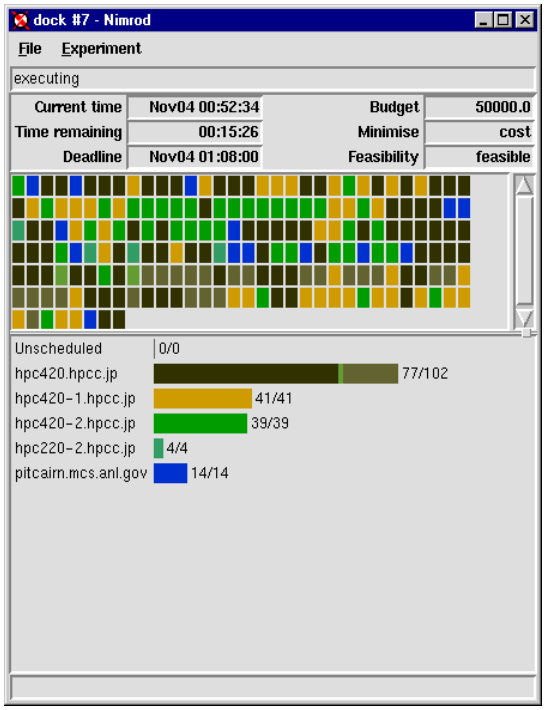


Figure 14: A snapshot of the Nimrod-G monitor during "Optimize for Cost" scheduling.

It should be noted that for each job scheduled for execution on the Grid, the Nimrod-G runtime machinery (actuator) deploys Nimrod-G agents on remote resources. The Nimrod agents setup runtime environments (generally in scratch file space, "/tmp") on remote resources and execute commands specified in the task definition script (see Figure 10). The docking parameter files and ligand specific files are transferred from the home node, bezek.dstc.monash.edu.au in this case. The agent uses *http* protocols to fetch files via the http-based file server running on the home node. All parameter variables in the parameterized input files (see Figure 8 and Figure 9) are substituted by their concrete values before processing. The ligand record is fetched from the CDB database server running on the home node. The agent then executes the *dock* program and stores output files in the scratch area. The required output files are then transferred to the home node and stored with the job number as their extension. All these steps involved in the execution of the *dock* program on Grid resources were completely hidden from the user.

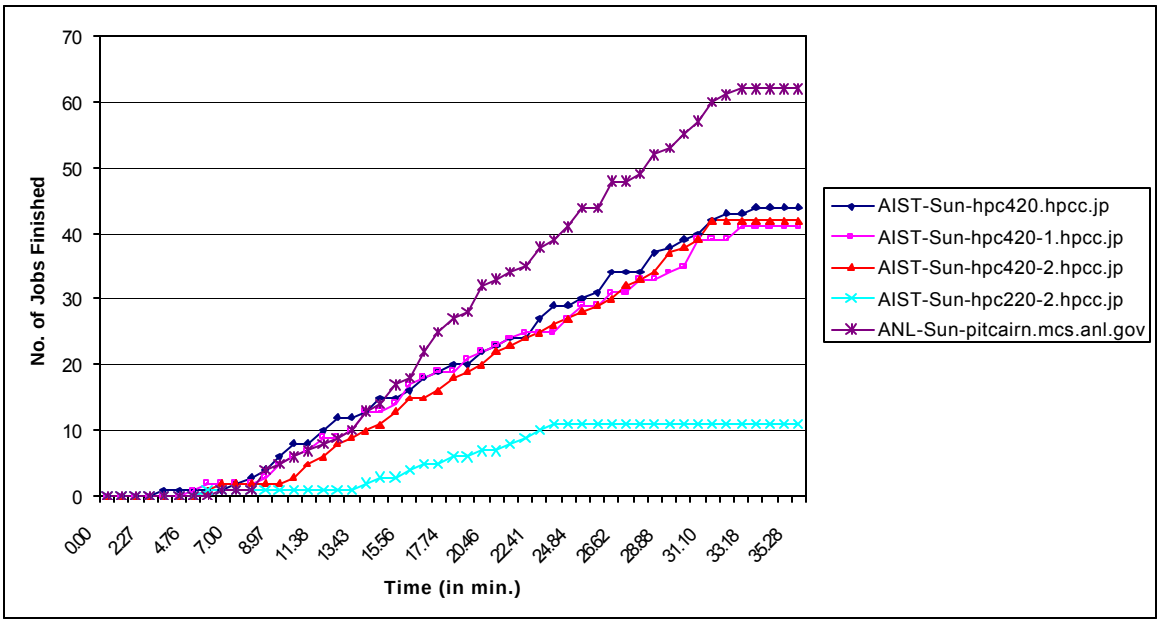


Figure 15: No. of jobs processed on Grid resources during DBC time optimization scheduling.

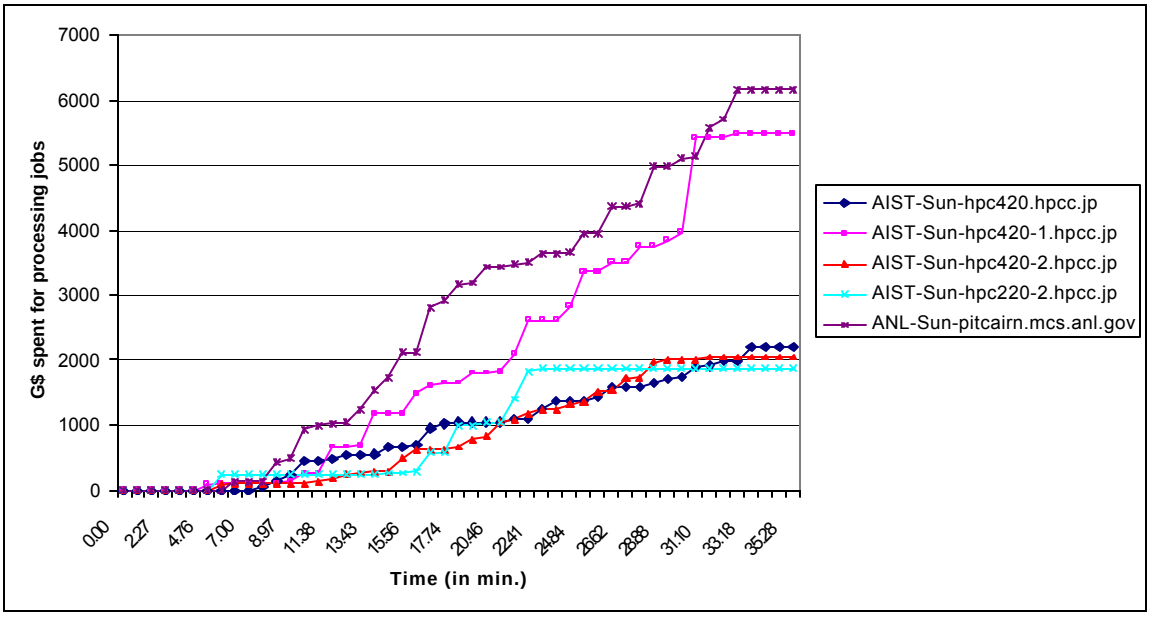


Figure 16: The amount spe nt on resources during DBC time optimization scheduling.

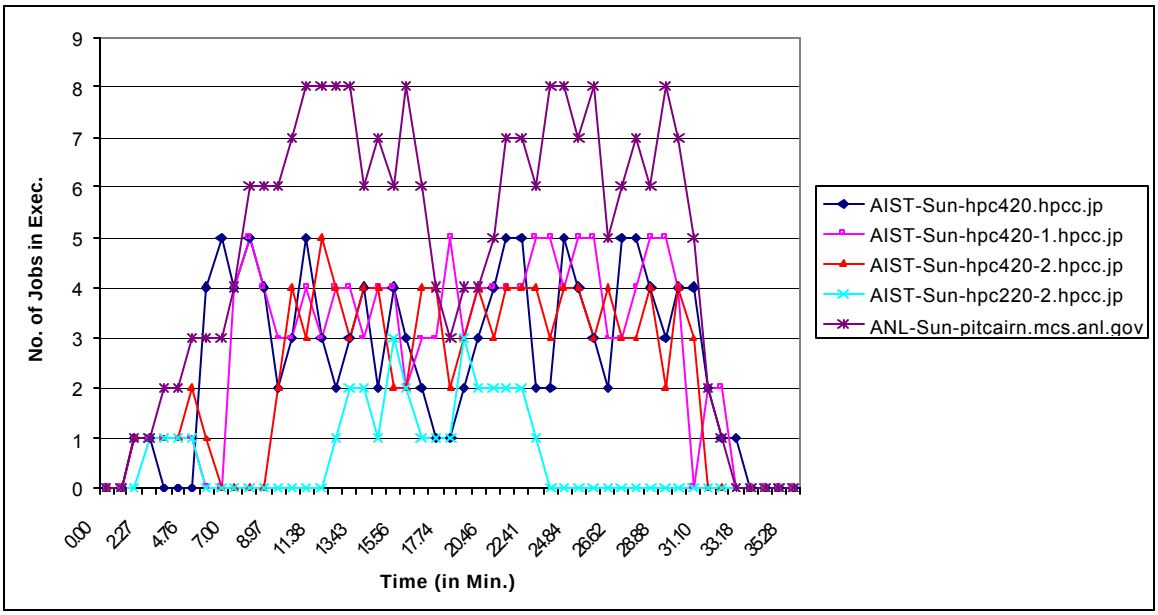


Figure 17: No. of jobs in execution on Grid resources during DBC time optimization scheduling.

The second experiment, *Optimize for Cost* scheduling, was performed on November 4, 2001 at 00:08:00, AEST, with a 60-minute deadline and finished on November 4, 2001 by 01:07:30. A snapshot of the Nimrod-G monitoring and steering client taken few minutes (~5min.) before the completion of application processing is shown in Figure 14. This experiment took almost 59.30 minutes to finish the processing of all jobs using resources available at that time with an expense of 14,277 G$. It is interesting to note that the second experiment took an extra 25.30 minutes and saved 3,425 G$ in the process. Figure 18 shows the number of jobs processed on different resources selected depending on their cost and availability. Figure 19 shows the corresponding expenses of processing on resources. Figure 20 shows the number of jobs in execution on resources at different times. From the graphs it can be observed that the broker selected the cheapest resources to ensure that the experiment was completed with minimum expenses, but before the deadline limit. In the beginning expensive resources are used to ensure that the deadline can be met. If for any reason cheapest resources are unable to deliver expected performance, then the broker seeks the help of expensive resources to meet the deadline.

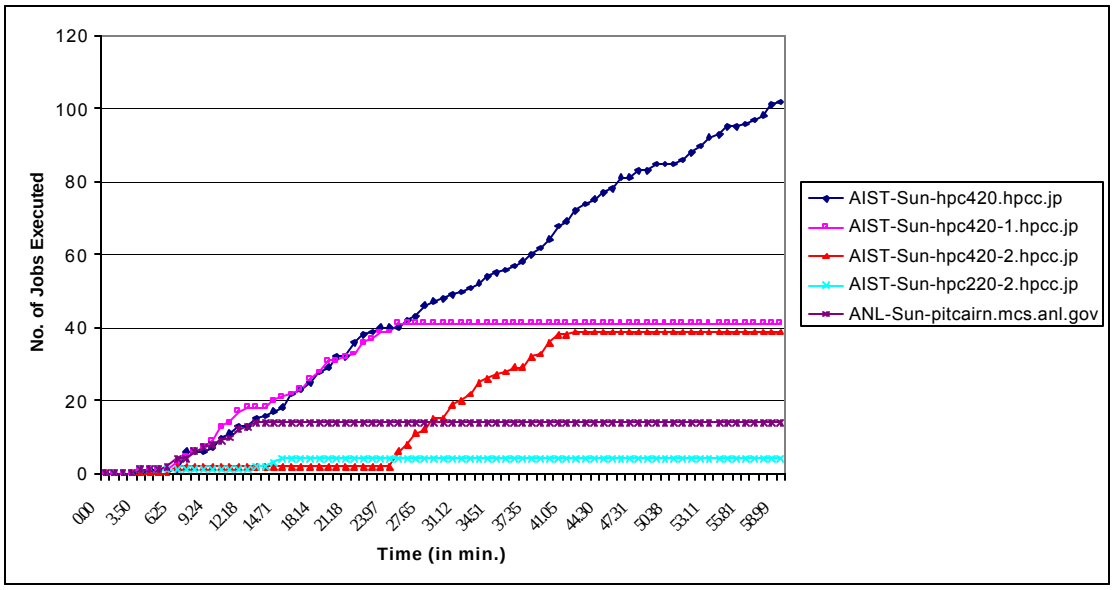


Figure 18: No. of jobs processed on Grid resources during DBC Cost optimization scheduling.

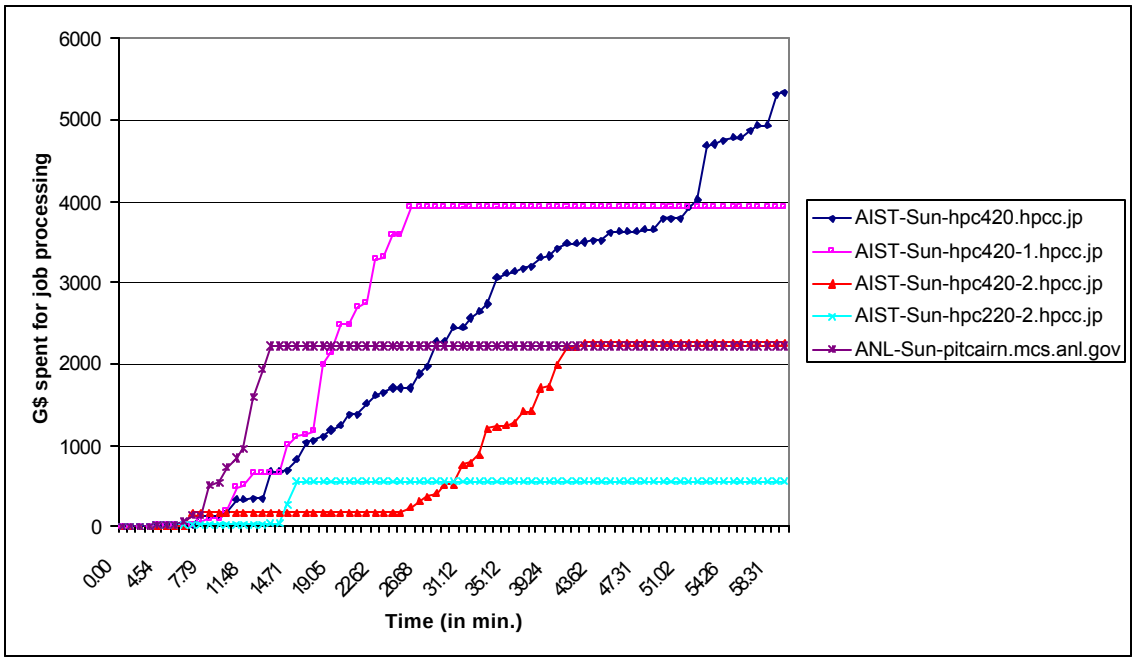


Figure 19: The amount spent on resources during DBC Cost optimization scheduling.

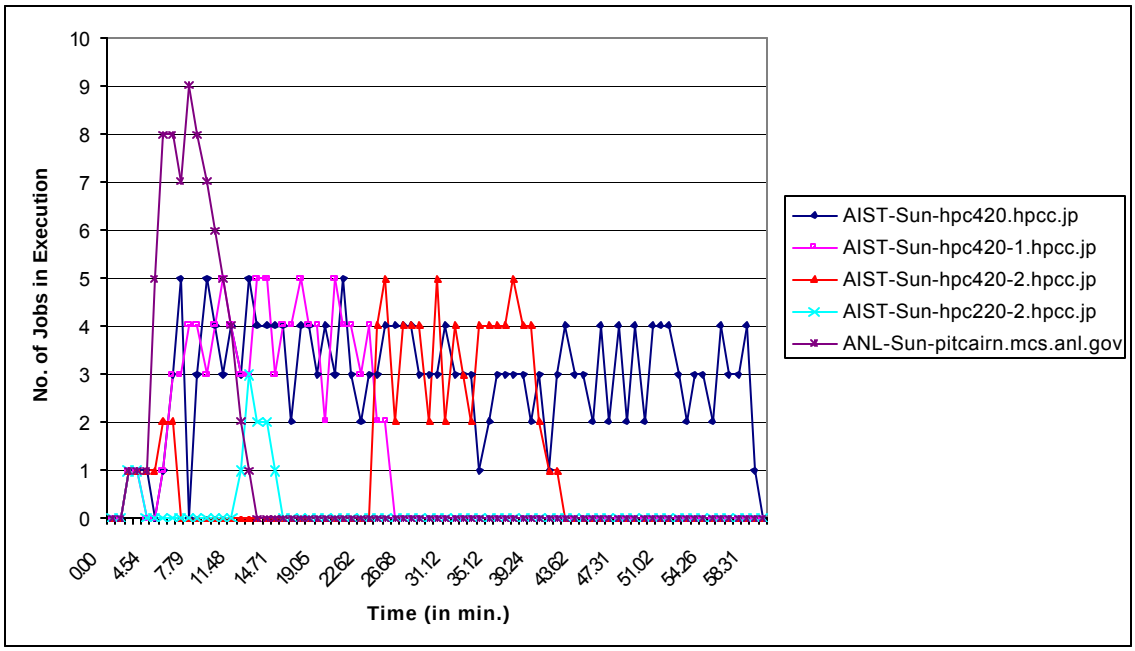


Figure 20: No. of jobs in execution on Grid resources during DBC Cost optimization scheduling.

6 Related Work

Although many researchers have explored the use of parallel computing techniques in molecular docking for drug design [22][23], there are only a few efforts that use the world wide distributed computers for processing docking jobs in parallel. One of the most related efforts is the FightAIDS@Home project [21], which is based on the Entropia's distributed computing network and the Scripps Research Institute's docking application. In this case, volunteers need to download Entropia's screen saver program that runs in the background on the volunteer computer. The volunteer PC contacts the Entropia server to download the

data to perform docking. When docking on an assigned data is completed, it uploads the results to the server. This execution model is different from our model where the scheduler (Nimrod-G) assigns the work to computers that are available and initiates the execution.

Most of the efforts explicitly develop docking application as a parallel application using a special purpose, legacy or standard, parallel programming languages and interfaces such as PVM and MPI, which requires extra development effort and time. The scalability of such applications and runtime systems is limited to resources available in a single domain and they need powerful computers and networks for faster processing.

Our techniques are novel in many ways. To perform parallel and distributed docking using our tools, there is no need to develop docking application as a parallel application. Our framework supports the composition of the existing molecular docking application as a parameter sweep application without making any changes to it. Our runtime machinery, the Nimrod-G resource broker, creates independent docking jobs automatically and launches their parallel execution on world-wide distributed computers. It hides all the complexity associated with scheduling jobs, shipping appropriated input files, starting and monitoring their execution, and shipping results back to the user. Our scheduler also supports the deadline and budget based scheduling, which prioritizes the processing depending on the user requirements—how quickly they need results, how much they want to spend, and which one to optimise.

7 Summary and Conclusion

Computational Grids enable the *sharing* and *aggregation* of geographically distributed resources for solving large-scale, resource and data-intensive problems faster and cheaper. However, application development, resource management, and scheduling in these environments is a complex undertaking. We have developed a Virtual Laboratory environment and tools for formulating molecular docking for drug design as a parameter sweep application, chemical database management, and scheduling docking jobs for processing on a wide area distributed resources by leveraging existing Grid technologies. The new tools developed include a chemical database indexer, CDB server for providing access to molecules in chemical databases as a network service, clients for accessing CDB services from a selected CDB service. We have used the Nimrod-G parameter specification language for composing an existing docking application as a parameter sweep application and the Nimrod-G grid resource broker for processing molecular docking jobs on distributed resources.

We have conducted deadline and budget constrained scheduling experiments for parallel processing of docking jobs on the worldwide grid testbed under two different optimization scenarios. The results of this molecular docking application scheduling on a large-scale distributed resources demonstrate the potential of the Virtual Laboratory tools for *service-oriented computing*. They prove the effectiveness of computational economy and quality of services (QoS) driven scheduling as an efficient mechanism for the management of supply-and-demand for resources depending on the value delivered to the user. The economy driven service oriented computing encourages the users to utilize resources effectively by trading off between the deadline and budget depending on their QoS requirements.

Our experience with developing a prototype Virtual Laboratory environment for distributed drug design shows the potential and applicability of the Nimrod-G tools for data intensive computing. We are extending the current system to support adaptive mechanisms for the selection of the best CDB service depending on access speed and cost. We are also looking into applying the experience gained in this work to develop a virtual laboratory environment for enabling high-energy physics events processing on distributed resources on a larger scale.

Acknowledgements

We would like to thank John Crossley (Monash U.) and Rob Gray (DSTC) for proof reading earlier drafts. They offered constructive comments and suggestions for enhancing the quality of the paper. We appreciate the efforts of Heinz Stockinger (CERN) for reviewing the chemical database server implementation. We thank National Institute of Advanced Industrial Science and Technology, Tokyo, Japan and the Argonne National Laboratory, Chicago, USA for providing access to the resources used in the experimentation. We thank Geoffrey Fox for handling reviews in record time. We express our special thanks to all six anonymous reviewers for their critical comments and excellent advice on improving the paper.

References

[1] I. Foster and C. Kesselman (editors), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, USA, 1999.

[2] R. Buyya (editor), *High Performance Cluster Computing: Architectures and Systems*, Volumes 1 and 2, Prentice Hall, NJ, USA, 1999.

[3] D. Abramson, R. Sosic, J. Giddy, and B. Hall, *Nimrod: A Tool for Performing Parameterized Simulations using Distributed Workstations*, The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[4] D. Abramson, J. Giddy, and L. Kotler, *High Performance Parametric Modelling with Nimrod/G: Killer Application for the Global Grid?* International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society Press, 2000.

[5] R. Buyya, D. Abramson, and J. Giddy, *Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid*, 4th Intl. Conf. on High Performance Computing in Asia-Pacific Region (HPC Asia 2000), China.

[6] R. Buyya, D. Abramson, and J. Giddy, *Economy Driven Resource Management Architecture for Computational Power Grids*, Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), USA.

[7] R. Buyya, J. Giddy, and D. Abramson, *An Evaluation of Economy-based Resource Trading and Scheduling on Computational Power Grids for Parameter Sweep Applications*, The Second Workshop on Active Middleware Services (AMS 2000), In conjunction with HPDC 2001, August 1, 2000, Pittsburgh, USA (Kluwer Academic Press).

[8] R. Buyya, H. Stockinger, J. Giddy, and D. Abramson, *Economic Models for Management of Resources in Peer-to-Peer and Grid Computing*, SPIE International Conference on Commercial Applications for High-Performance Computing, August 20-24, 2001, Denver, USA.

[9] I. Foster and C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputer Applications*, 11(2): 115-128, 1997.

[10] T. Ewing (editor), *DOCK Version 4.0 Reference Manual*, University of California at San Francisco (UCSF), USA, 1998. Online version: http://www.cmpharm.ucsf.edu/kuntz/dock.html

[11] E. Lunney, *Computing in Drug Discovery: The Design Phase*, IEEE Computing in Science and Engineering Magazine, http://computer.org/cise/homepage/2001/05Ind/05ind.htm

[12] Tripos, Inc., *SYBYL Mol2 File Format*, http://www.tripos.com/services/mol2/, USA, Oct. 7, 2001.

[13] Tripos, Inc., *SYBYL Toolkit for molecular design and analysis*, http://www.tripos.com/software/sybyl.html, USA, Oct. 7, 2001.

[14] I. Kuntz, J. Blaney, S. Oatley, R. Langridge, and T. Ferrin, *A geometric approach to macromolecule-ligand interactions*, Journal of Molecular Biology, 161: 269-288, 1982.

[15] B. Shoichet, D. Bodian, and I. Kuntz, *Molecular docking using shape descriptors*, Journal of Computational Chemistry, 13(3): 380-397, 1992.

[16] E. Meng, D. Gschwend, J. Blaney, and I. Kuntz, *Orientational sampling and rigid-body minimization in molecular docking*, Proteins, 17(3): 266-278, 1993.

[17] TurboLinux, *EnFuzion Manual*, http://www.turbolinux.com/downloads/enf/man/enfuzion.htm

[18] R. Buyya, *World Wide Grid (WWG)*, http://www.csse.monash.edu.au/~rajkumar/ecogrid/wwg/

[19] W. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, D. Anderson, *A new major SETI project based on Project Serendip data and 100,000 personal computers*, Astronomical and Biochemical Origins and the Search for Life in the Universe—Proceedings of the Fifth International Conference on Bioastronomy, 1997.

[20] R. Buyya, *The Virtual Lab Tools for Chemical Database (CDB) Management and Access*, http://www.buyya.com/vlab/

[21] Entropia, *Fight AIDS at Home Project: A joint effort of Entropia and Scripps Research Institute*, http://www.fightAIDSatHome.org/

[22] G. Morris, D. Goodsell, R. Huey, A. Olson, *Distributed automated docking of flexible ligands to proteins: Parallel applications of AutoDock 2.4*, Journal of Computer-Aided Molecular Design, Vol. 10, pp. 293-304., 1996.

[23] R. Schneider, *Drug Design Using Parallel Processor*, http://www.cordis.lu/esprit/src/results/res_area/hpnc/hpnc15.htm

[24] J. Brooke, M. Foster, S. Pickles, K. Taylor, and T. Hewitt, *Mini-Grids: Effective Test-beds for Grid Application*, Proceedings of the First IEEE/ACM International Workshop on Grid Computing (GRID 2000), Dec. 17, 2000, Bangalore, India: Springer Verlag Press, Germany.

[25] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, *Economic Models for Resource Management and Scheduling in Grid Computing*, The Journal of Concurrency and Computation: Practice and Experience (CCPE), Wiley Press, USA, May 2002.

[26] R. Buyya, *Economic-based Distributed Resource Management and Scheduling for Grid Computing*, Ph.D. Thesis, Monash University, Melbourne, Australia, April 12, 2002. Submitted version is Online at: http://www.buyya.com/thesis/thesis.pdf

A Flexible Grid Framework forAutomatic Protein-Ligand Docking

§ David Abramson, † Celine Amoreira, † Kim K. Baldridge,
‡ Laura Berstis, ‡ Chris Kondrick, § Tom Peachey
davida@csse.monash.edu.au, kimb@oci.unizh.ch

§ *CSIT, Monash University, Clayton, 3800, Victoria, Australia*

† *Faculty of Chemistry, University of Zurich, Irchel, Zurich, Switzerland*

‡ *San Diego Supercomputing Center, 9500 Gilman Drive, MC 0505 La Jolla, CA 92093-0505, USA*

Abstract

Many important and fundamental questions in biology and biochemistry can be better understood through investigations performed at the protein-ligand or drug-receptor level. A variety of techniques have been used over the years, and it is an area of active research. In this paper we illustrate an approach that leverages a number of different computational chemistry approaches, and combines these with non-linear optimization algorithms and grid based high performance computing platforms. The result is a very flexible, high performance method of evaluating protein-ligand interaction algorithms. We illustrate the approach by evaluating a hybrid molecular modeling and quantum theoretical based algorithm.

1. Introduction

Many important and fundamental questions in biology and biochemistry can be better understood through investigations performed at the protein-ligand or drug-receptor level. Experimental research questions involving macromolecular biological complexes are typically directed at understanding the correlation between molecular and structural specialization and functional variations. However, typically, it becomes very difficult to determine the exact mechanism of action at the molecular level without the aid of computational investigations at the atomic level. Additionally, estimating correct three-dimensional atomic structures of complexes between proteins and ligands is an important component of the drug-design process in the pharmaceutical industry. Basic aspects of ligand-protein interactions, categorized under the general term molecular recognition, are concerned with the specificity as well as the stability of the ligand to bind the binding site. To this end, molecular recognition is central to the development of active substances that could be used as drugs, or designing ligands that can undergo specialized function.

Molecular recognition can be studied by looking at the properties and mechanistic of binding between the protein and a ligand, in the biological environment. Investigations involved in the positioning of the ligand into the macromolecule can be performed very specifically in the case that the researcher is interested in understanding the variance of binding energy as a ligand moves in very specific ways (e.g., around specified dihedral angles), or, they may be interested in probing all degrees of freedom of motion of the ligand in the protein pocket. There has been considerable effort in studies of protein-ligand interactions and many tools and software have been developed. However, much still remains to be done as no single generalized tool has provided a truly reliable and accurate method for understanding detailed ligand-protein interactions. Computational models that perform these calculations typically have significant performance demands, and thus high performance parallel and distributed computers can be desirable.

Rather than equip a single tool to perform all of the functions, we favour a framework in which key tools can be combined. In the area of protein-ligand docking, one benefits from separating the process of computing molecular interaction from a chemical viewpoint, from the non-linear optimization that is performed in order to minimize the binding energy. Moreover, utilization of Grid-based distributed supercomputers can be difficult, and the computational strategies that are required can make software that is already complex even more so.

Our approach for this type of problem involves the coupling of a number of separate codes, each focused on a particular aspect of the total problem, using Grid-oriented strategies. In our case, the chemistry is performed via a combination of quantum chemistry and classical molecular mechanics methods, the non-linear optimization is performed using the package called Nimrod/O [1][3], and Grid based high performance computing is performed with a specialized tool called Nimrod/G [2][4].

In this paper we discuss the approach we have used. Section 2 discusses classical approaches to modeling the protein-ligand docking process. Section 3 describes a new experimental technique that we have been evaluating. Section 4 introduces a framework that leverages the Nimrod family of software tools. In Section 5 we illustrate the results of a case study.

2. Classical approaches to protein-ligand docking

Due to the impact on human health (drug design), huge efforts are devoted to research in areas involving details of protein-ligand interaction, including the development of reliable software tools. The primary objective in ligand-protein interaction investigations is the prediction of the optimal orientation and conformation of a small molecule (e.g., ligand, drug prototype, etc) in a pocket embedded in a protein. Such investigations can be carried out using a variety of different methods, depending on the questions being asked.

Computational methods for the study of ligand-protein interactions vary considerably in their algorithmic structure and resource requirements, as shown in Figure 1. Quantum mechanical (QM) based methods that operate at the electronic structure level are relatively limiting in terms of molecular size and computational time, but provide high level of accuracy. Classical-based methods use less accurate strategies that neglect the electronic motion, but only require modest compute time. Molecular dynamics and Monte Carlo, both molecular mechanics (MM) based approaches, involve long simulation times but are effective within the classical limits. Molecular docking procedures, widely used for study of ligand-protein interactions, place a ligand in a receptor using local shape features and electrostatic criteria based on a number of different models.

Ab initio QM is the holy grail for highly accurate results for molecular systems and is based on algorithms that solve the Schrödinger equation. Unfortunately, most problems are too large to be considered fully by QM methods alone. Therefore, approximations that consider less electronic structure detail are incorporated, often emphasizing molecular energy as a function of the nuclear positions only.

The MM-based methods are fully empirical in nature, and are specifically tuned for classes of molecules such as proteins, enzymes, or other macromolecular systems. In MM, molecules are considered collections of masses held together by classical forces. The equations based on classical mechanics and parameters defining the energy surface of a molecule are collectively referred to as force field. The contributions to the molecular energy include, bond stretching, angle bending dihedral deformations, van der Waals, electrostatic interactions, etc. An important issue for force field applications is the feasibility for transferability of the model to a wider range of macromolecules. The MM method is largely applied for basic understanding of low energy conformations on the Potential Energy Surface (PES) through molecular structure refinement, molecular dynamics (MD) simulation, Monte Carlo simulations and/or ligand docking simulation.

Molecular Dynamics, based on the empirical force field and basic assumptions described for MM, aims to reproduce the time-dependent motion of a molecule by solving the Newton's second law for all the atomic degrees of freedom. New positions and velocities of the atoms are predicted and saved as a trajectory file. Performing this process for a range of time steps with additional controls on temperature gives the description of the behavior of the molecular motion as it traverses over the PES. In contrast to QM and MM, both of which attempt to reach the minimum in the energy function, molecular dynamics is able to overcome conformational barriers with the input of temperature. The ability for Molecular Dynamics to search conformational space enables the placement of families of structure on the PES, which can be further minimized for prediction of local stationary points.

Current molecular docking tools attempt to estimate the optimal three-dimensional configurations between a protein and a ligand using a variety of classical means, for example, by invoking an empirical force field function. Typically, docking searches involve a large number of ligands and associated conformations, together with a cost or scoring function, which governs the degree of compatibility between the ligand and the protein in the formation of a reaction complex.

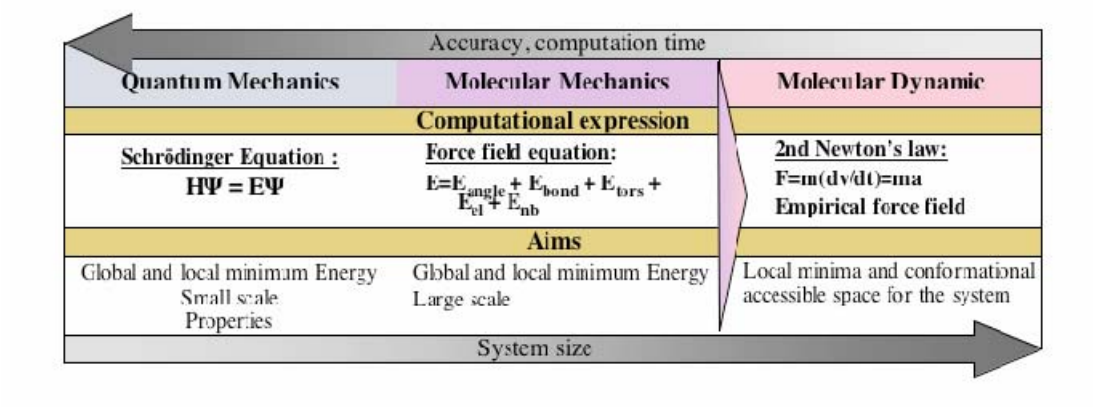


Figure 1: Computational methods for study of protein-ligand interactions

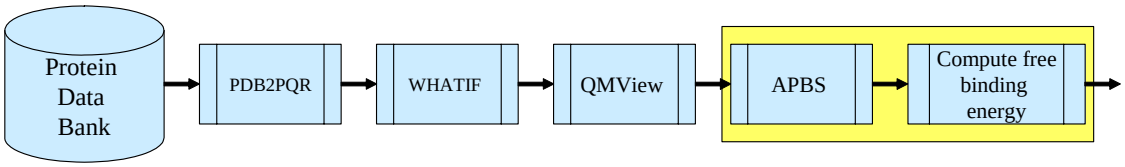


Figure 2: Hybrid protein-ligand docking pipeline

3. A hybrid docking algorithm

Recently, we proposed a new hybrid algorithm that uses both QM and classical methods, and this forms the basis for the experiments we report later in the paper. The overall technique could equally be applied to any of the methods discussed in the previous section. Typically, investigations begin with a structure from the Protein Data Bank (PDB), [9] which provides experimental structures of ligands, proteins, or enzymes. These crystal structures are without hydrogen atoms, which must be added using tools such as Babel or PDB2PQR [7]. In the case that there is a water network attached to the protein, it might be necessary to remove the network, with tools such as WHATIF [8]. After this phase of preparation, a checking point is needed to verify the protonation state of the ligand, and optimal positioning of the hydrogen atoms.

After the initial structures are prepared, the next step is to consider the possible positioning of the ligand within the protein pocket. The researcher may be interested in the variance of binding energy as a ligand moves around only specified degrees of freedom, or, they may be interested in probing all degrees of freedom of motion of the ligand in the pocket. We have a tool, QMView, [5], that we have enhanced to enable the generation of ligand conformations resulting from any particular metric, for example rotation around a torsional angle, and subsequent placement into the protein by aligning the new conformation with a part of the ligand that remains fixed. A second checking point is then required to delete the non-stable conformations due to unnatural bumping between atoms.

In the various steps, the electrostatic effects can be investigated using a wide variety of available tools. As opposed to using classical force fields for this, we focus on a package that solves the Poisson-Boltzmann equation (PBE) numerically (e.g., APBS [12]). This method describes the protein and ligand as collections of charges and radii within a continuous solution of specified dielectric, salt concentration and temperature. The atomic charges and radii for the ligand, and optionally various levels of structure, are determined using more sophisticated QM methodologies such as the CHELPG method [6], using the ab initio molecular QM program GAMESS [13]. Additionally, QM methods can provide more accurate structural information if need be as well as an estimate of internal energy assessment,. The protein charges and radii must also be specified for the electrostatics computation, and are obtained for example from the PARSE [14] parameterization, as they are well defined for estimating the contribution of simple functional groups to solvation.

After solving the PBE, APBS computes the binding free energy, which is used as an objective function value to drive the conformational search. All of the tools discussed to date can be combined in a variety of

ways, ranging from simple shell or Perl scripts, through to complete workflows managed by one of a number of workflow tools, such as Kepler [10][11].

The output of this pipeline, as shown in Figure 2, is the binding free energy for a *particular* protein-ligand configuration. In order to actually minimize the binding free energy, we need to adjust the configuration of the ligand within the protein binding site. In several docking codes, this non-linear optimization phase is embedded in the chemistry code itself, for example, AutoDock [15]. While this is powerful, it is not possible to experiment with different search procedures or different types of chemical calculations without modifying a significant amount of the software package. In this work, just as we are using several different chemistry packages to compute the binding energy, can also combine these with a state of the art nonlinear optimization package to perform searches and other parameterization procedures, the topic of discussion in the next section.

4. A docking framework

As discussed in section 1, our strategy involves separation of three computational aspects of the molecular docking algorithm, namely

- the computational chemistry that computes the binding energy;
- the optimization of the ligand position within the protein and
- the use of high performance computational platforms.

Figure 3 shows a framework that supports this approach using a range of software tools. At the core is the particular protein-ligand docking algorithm, such as the one discussed in the last section. The other two components of the framework are served by two members of our Nimrod tool family, namely Nimrod/O [1][3] and Nimrod/G [2][4].

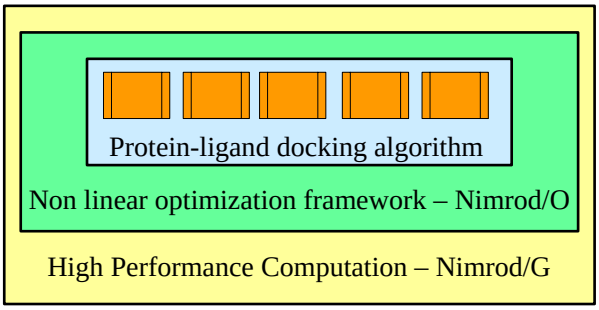


Figure 3 – Docking Framework

Nimrod/O is a generic tool that performs distributed non-linear optimization. This tool can be applied to any problem in which an objective cost function is computed by a computational model, and thus, is an ideal tool to use in performing protein-ligand docking as described in Section 3. Nimrod/O is optimized for computer models that are computationally expensive. Numerical optimization using these models typically needs multiple iterations of the search heuristic, each requiring several runs of the model.

The objective landscape produced by computational models frequently shows multiple local optima, and the binding energy surfaces explored for protein-ligand interactions are no exception. In addition, discretization errors introduced by these models may introduce many more local optima as artifacts. The local optimum found by a search algorithm will depend on the algorithm used and often on the starting point chosen for the search. Accordingly, Nimrod/O allows multiple concurrent searches from a range of starting points. The tool also allows many of the different optimization methods discussed above to be executed concurrently. If all optimizations converge to the same optimum, then the user can be confident that this is a global optimum. If all optimizations do not converge to the same point, then the range of local optima is an indication of the roughness of the landscape and serves to guide further searches.

```
parameter x float range from -2 to 2;
parameter y float range from -4 to 4;
parameter z float range from -4 to 4;
parameter xR float range from -3.15 to 3.15;
parameter yR float range from -3.15 to 3.15;
parameter zR float range from -3.15 to 3.15;
```

```
task main
   node:execute transrad $x $y $z $xR $yR $zR
   copy root:skeleton   node:.
   node:substitute  skeleton   apbs.input
   node:execute apbs  apbs.input   > apbs.output
   node:execute extract apbs.output > output.$jobname
   copy node:output.$jobname  root:.
endtask
```

```
method simplex
  starts 10 named "simplex"
    starting points random
    tolerance 0.0
  endstarts
endmethod
```

```
method bfgs
  starts 10 named "bfgs"
    starting points random
    tolerance 0.001
  endstarts
endmethod
```

Figure 4: A Nimrod/O schedule file for protein-ligand docking.

In general, a Nimrod/O experiment is performed by preparing a declarative text file called a "schedule". Figure 4 shows a schedule for performing protein-ligand docking by minimizing the binding energy function, using a number of the packages discussed in Section 3.

The task of minimizing the binding energy as a function of ligand position in the protein pocket is a search within a six dimensional parameter space. Accordingly, six parameters (x, y, z, xR, yR, zR) indicating the position and rotation in three dimensional space are defined in the schedule using the "parameter" statements. The "task" section specifies the operations required to perform one instance of the computational model and produce a numerical output that will be the objective function for the optimization. First, the program "transrad" translates and rotates the ligand, aborting if a collision occurs between the protein and the ligand. Then a "skeleton" file is copied from the root node to the remote computational node where execution will occur. The subsequent "substitute" command edits the skeleton file, replacing the parameters by their current values, producing a file "apbs.input". Then the program "apbs" acts on the input file producing a file "apbs.output". This output file is finally filtered to produce the numerical value for the binding energy that is copied back to the root node. The final section of the schedule file specifies the optimization method to be used. In this example, two well-known algorithms are invoked: the downhill simplex and the BFGS gradient descent method. Ten starts are performed with each, giving twenty distinct optimizations.

To improve the performance of the system, batches of jobs are run in parallel using Grid resources. To do this, Nimrod/O uses parallel versions of standard search methods, for example, gradient descent (BFGS), simplex, simulated annealing and evolutionary techniques such as genetic algorithms. The tool also uses a range of computational resources including uni-processor workstations, parallel clusters and distributed super-computers. Nimrod/O does this by leveraging a related tool, Nimrod/G to perform the distribution of computational tasks over a Grid of resources [2][4].

Nimrod/G is used to explore the behaviour of complex systems, modelled by existing computational software packages, as the input conditions change, enabling users to design better procedures, understand complex processes, and/or optimise system behaviour. Nimrod/G incorporates a distributed scheduling component that manages the scheduling of individual experiments to a wide range of computational resources, including workstations, clusters and high-end supercomputers. The tool sits on top of conventional schedulers like PBS [16] and Condor [17], as well as Grid middleware such as Globus [18]. Together, these features assure that even complex parametric experiments can be defined and run with little programmer effort. In many cases it is possible to establish a new experiment in minutes.

5. A case study

In this section, we show the results of a number of computational experiments as outlined in the previous section for a specific protein and ligand system, for the evaluation of the binding free energy using the APBS software. Three non-linear optimization techniques were selected: simplex, gradient descent (BFGS) and Simulated Annealing (SA). A total of 20 Simplex, 20 BFGS and 10 different SA configurations were executed to explore the behaviour of the different algorithms.

5.1. Using a local workstation & local cluster

As discussed in the last section, Nimrod/G is flexible enough to perform the computations on anything from a local workstation, a cluster, or a set of distributed clusters. In this section we report the performance of the experiment on a local workstation and a small cluster. In the first case, the optimization runs are executed sequentially on a uni-processor workstation. In the second case, each optimization algorithms perform a number of functional evaluations in parallel using a cluster. However, the runs are then executed sequentially since there are insufficient resources in the cluster to run them all in parallel.

The results of these simulations are summarized in Table 1. We note that, not all of the simulations were able to complete and find a local optimum. For example, of the 20 Simplex runs, 10 failed to complete because the energy calculation failed for part of the search space. Only 5 BFGS and 5 SA simulations completed. This is not uncommon in the use of computational models that may not be defined in all regions of the search space. In spite of this, a total of 20 different starting points were evaluated, yielding a range of solutions. as summarized in Table 1. "Run#" identifies the particular run. "Optimum" reports the final binding energy. In these experiments, only the APBS electrostatic computation is evaluated. "Iters" shows how many iterations of the heuristic were

performed, whereas “Evals” shows how many times the binding energy was computed. “Batches” reports the number of parallel batches of evaluations, which is a good indication of the variability in wall clock time since this indicates the critical path in the search. “Cluster Time” shows the effect of running each batch in parallel, and is the sum of the batch times for each optimization. “Workstation Execution Time” reports the time that the jobs took when executed sequentially on a uni-processor workstation.

The best result was obtained from one of the SA runs (SA Run 1), and the worst result from one of the Simplex runs Simplex Run 1), however, there is a wide variability and no single heuristic performed uniformly poorly. Moreover, each of the runs took a different number of iterations of the search heuristics, which in turn caused a varying number of objective function evaluations and parallel batches. It is worth observing that, while SA produced the best result, this method also required the longest time to complete.

5.1. Using the grid

To demonstrate the ability to run the experiments on the Grid, all these optimizations were repeated in parallel. This time, Nimrod/G was configured with a small Grid testbed, and shown in Table 2. On all these resources, the experiment was competing for CPUs with other users, and thus even though a given resource may have had many processors, we were not able to utilize all of them at the same time. Figure 5 shows how the number CPUs used varied as the experiment progressed. As optimizations complete, or fail, the resource requirements attenuate.

The results of this case study indicate that the overall approach is viable – clearly it is possible to couple the many different packages that we used to successfully explore different protein-ligand bindings. By using this Grid, we were able to run the experiment some 33 times faster than the workstation, and about 9 times faster than the small cluster.

6. Conclusions

In this paper we have reviewed a range of techniques that can be used to explore the dynamics of protein-ligand binding, and have evaluated a particular approach involving a hybrid between classical molecular dynamics and quantum theoretic approaches.

A major problem in docking is the determination of a realistic energy cost function. This function is usually assumed to consist of an electrostatic energy contribution, a nonelectrostatic energy and entropic contribution, and an internal energy part. The electrostatic binding energy between a protein and a ligand can be relatively accurately computed by dielectric continuum solvation methods based on the Poisson-Boltzmann equation. Since in contrast to the protein, the charge distribution on the ligand is usually unknown, quantum chemical methods can be used. This provides more accurate structure, charge information, and estimation of internal energy. The non-electrostatic and entropic contributions are fit to experimental results and included. Future goals involve exploration of more complex techniques to produce more accurate models of the protein-ligand interaction in a general procedure.

The approach presented here uses a range of different software packages for study of ligand/protein interactions, enabling prediction of binding energy for a ligand docking into a binding site. Further, because we separated the non-linear optimization component from the chemistry evaluation component, we are able to ultilize a Grid enabled package to perform the latter task. Overall, this generates considerable flexibility and will allow us to explore a range of modeling techniques in the future.

Acknowledgements

The projects are supported by a variety of funding agencies, including the Australian Research Council, the US National Science Foundation, the Swiss National Science Foundation, and PRAGMA.

References

[1] Abramson, D, Lewis A, Peachey T, Fletcher, C., “An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics”, SuperComputing 2001, Denver, Nov 2001.

[2] Abramson, D., Sosic R., Giddy J. and Hall B., “Nimrod: A Tool for Performing Parametrised Simulations using Distributed Workstations”, The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.

[3] Abramson, D, Lewis, A. and Peachy, T., “Nimrod/O: A Tool for Automatic Design Optimization”, The 4th International Conference on Algorithms & Architectures for Parallel Processing (ICA3PP 2000), Hong Kong, 11 - 13 December 2000.

[4] Abramson, D., Giddy, J. and Kotler, L. “High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?”, International Parallel and Distributed Processing Symposium (IPDPS), Cancun, Mexico, May 2000, pp 520- 528.

[5] Baldridge, K.K.; Greenberg, J.P. QMView: “A

Computational 3-D Visualization Tool at the Interface Between Molecules and Man". *J. Mol. Graphics* 1995, *13*, pp 63-66.
[6] Breneman, C.M., Wiberg K.B., "Determining atom-centered monopoles from molecular electrostatic potentials", *J. Comp. Chem.*, 1990, *11*, pp 361-373.
[7] Dolinsky, TJ, Nielsen JE, McCammon JA, Baker NA. "PDB2PQR: an automated pipeline for the setup, execution, and analysis of Poisson-Boltzmann electrostatics calculations". *Nucleic Acids Research*, 32 W665-W667 2004, http://pdb2pqr.sourceforge.net/
[8] Vriend, G., "WHAT IF: A molecular modeling and drug design program", *J. Mol. Graph 8*, 1990, pp 52-56, http://swift.cmbi.kun.nl/whatif/
[9] H.M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T.N. Bhat, H. Weissig, I.N. Shindyalov, P.E. Bourne, "The Protein Data Bank", *Nucleic Acids Research* , 28 pp. 235-242 (2000), http://www.rcsb.org/pdb/
[10] http://kepler-project.org
[11] Ludäscher, B. Altintas, I. Berkley, C. Higgins, D. Jaeger-Frank, E. Jones, M. Lee, E. Tao J. and Zhao Y., "Scientific Workflow Management and the Kepler System", *Concurrency and Computation: Practice & Experience*, Special Issue on Scientific Workflows, 2005.
[12] M. Holst and F. Saied, "Multigrid solution of the Poisson-Boltzmann equation", *J. Comp. Chem.* 14, 1993, pp 105-113, http://apbs.sourceforge.net/
[13] Schmidt, M.W., Baldridge, K.K., Boatz, J.A., Elbert, S.T., Gordon, M.S., Jensen, J.H., Koseki, S., Matsunaga, N., Nguyen, K.A., Su, S., Windus, T.L., "The General Atomic and Molecular Electronic Structure System" *J. Comp. Chem. 14*, 1993, pp 1347-1363.
[14] Sitkoff, D.; Sharp, K.A.; Honig, B. "Calculation of Hydration Free Energies Using Macroscopic Solvent Models", *J. Phys. Chem* , 98, 1994, pp 1978-1988.
[15] Morris, G. M., Goodsell, D. S., Halliday, R.S., Huey, R., Hart, W. E., Belew, R. K. and Olson, A. J. "Automated Docking Using a Lamarckian Genetic Algorithm and and Empirical Binding Free Energy Function", *J. Comp. Chem* 19, pp 1639-1662.
[16] http://www.openpbs.org/
[17] http://www.cs.wisc.edu/condor/
[18] http://www.globus.org/

Algorithm	Run#	Optimum	Iters	Evals	Batches	Cluster Execution Time (mins)	Workstation Execution Time (mins)
Simplex	1	**80.8**	27	114	29	1044	2089
Simplex	2	No result		7	1	2	14
Simplex	3	No result		7	1	2	14
Simplex	4	73	21	90	22	663	1729
Simplex	5	No result		7	1	2	14
Simplex	6	68.2	63	277	68	2364	6148
Simplex	7	No result		7	1	54	225
Simplex	8	68.4	65	265	67	3023	6301
Simplex	9	69.6	67	262	72	3077	6428
Simplex	10	No result		7	1	56	236
Simplex	11	No result		7	1	56	237
Simplex	12	69.3	67	259	72	2402	6265
Simplex	13	70.8	43	139	43	1052	2836
Simplex	14	No result		7	1	59	251
Simplex	15	No result		7	1	59	253
Simplex	16	70.3	67	244	69	2488	5856
Simplex	17	No result		7	1	60	259
Simplex	18	72.7	46	207	50	2179	5202
Simplex	19	No result		7	1	60	267
Simplex	20	70.5	41	146	43	1397	3441
BFGS	1	78.3	3	62	11	400	2202
BFGS	2	No result		1	1	2	2
BFGS	3	No result		1	1	2	2
BFGS	4	No result		13	2	259	1109
BFGS	5	No result		1	1	3	3
BFGS	6	No result		13	2	539	1662
BFGS	7	No result		1	1	1	1
BFGS	8	No result		13	2	537	2088
BFGS	9	75.1	3	55	9	733	3435
BFGS	10	No result		1	1	1	1
BFGS	11	No result		1	1	1	1
BFGS	12	70.9	7	173	28	968	4885

BFGS	13	73.3	7	161	26	1314	6508
BFGS	14	No result		1	1	1	1
BFGS	15	No result		1	1	1	1
BFGS	16	No result		1	1	2	2
BFGS	17	No result		1	1	2	2
BFGS	18	69.6	4	108	19	502	2818
BFGS	19	No result		1	1	2	2
BFGS	20	No result		13	2	1550	8846
SA	1	**66.5**	16	815	160	3705	12746
SA	2	No result		226	12	225	3874
SA	3	No result		221	12	228	3874
SA	4	72.5	3	338	40	1000	5699
SA	5	No result		218	12	228	3873
SA	6	67	14	725	142	3282	11227
SA	7	No result		225	12	228	3878
SA	8	68.5	4	341	50	1181	6325
SA	9	69	8	495	84	1992	8655
SA	10	No result		223	12	228	3878
Best		66.5	3	1	1	1	1
Worst		80.8	67	815	160	3705	12746
Ave		71.2	29	130	24	784	2913
SD		3.6	26	180	36	1035	3230
Total (mins)						39216	145665
Speedup on Grid						9	33

Table 1 –Experimental results

Name	Institution	Processor Speed (MIPS)	No. CPUs
mahar.infotech.monash.edu.au	Monash Univ., Australia	5977	50
tgc.trecc.org	NCSA, USA	3985	13
brecca-2.vpac.org	VPAC, Australia	5571	180
pragma001.grid.sinica.edu.tw	ASGC, Taiwan	4771	11

Table 2 –Computational resources

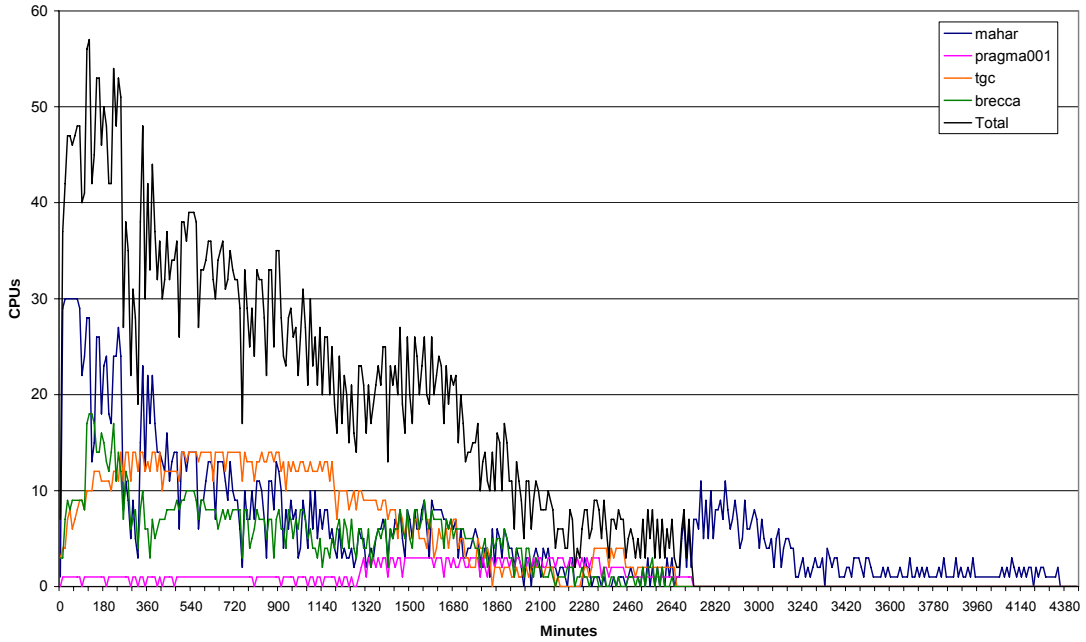


Figure 5 – Progress of jobs on grid testbed

Available online at www.sciencedirect.com
SCIENCE DIRECT®
Future Generation Computer Systems 21 (2005) 27–35
FGCS FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/future

Application of grid computing to parameter sweeps and optimizations in molecular modeling

Wibke Sudholt[a,*], Kim K. Baldridge[a,b], David Abramson[c], Colin Enticott[c], Slavisa Garic[c], Chris Kondric[b,c], Duy Nguyen[b,c]

[a] *Institute of Organic Chemistry, University of Zürich, 190 Winterthurerstrasse, Zürich 8057, Switzerland*
[b] *San Diego Supercomputer Center (SDSC), University of California, San Diego (UCSD), 9500 Gilman Drive, La Jolla, CA 92093-0505, USA*
[c] *Center for Enterprise Distributed Systems (DSTC), Monash University, Clayton, Victoria, 3800 Australia*

Available online 5 November 2004

Abstract

In science and engineering in general and in computational chemistry in particular, parameter sweeps and optimizations are of high importance. Such parametric modeling jobs are embarrassingly parallel and thus well suited for grid computing. The Nimrod toolkit significantly simplifies the utilization of computational grids for this kind of research by hiding the complex grid middleware, automating job distribution, and providing easy-to-use user interfaces. Here, we present examples for the usage of Nimrod in molecular modeling. In detail, we discuss the parameterization of a group difference pseudopotential (GDP). Other applications are protein–ligand docking and a high-throughput workflow infrastructure for computational chemistry.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Grid computing; Parameter sweep; Parameter optimization; Computational chemistry; Pseudopotential

1. Introduction

In science and engineering today, real as well as model systems tend to be complicated and highly parameterized. This can often create difficulties in the ability to predict the influence of each of the many parameters deterministically. Therefore, there is a large need for automated parameter sweeping and optimization procedures.

With all the exciting developments in the life sciences in the past few years, parametric modeling also has became more and more important in this field of research. Biological systems are intrinsically extremely complex. In addition, the number and character of the parameters is often not well known or understood, let alone their special influence on a biological structure.

* Corresponding author.
E-mail addresses: wibke@oci.unizh.ch (W. Sudholt), kimb@oci.unizh.ch (K.K. Baldridge), davida@csse.monash.edu.au (D. Abramson), Colin.Enticott@csse.monash.edu.au (C. Enticott), Salvisa.Garic@infotech.monash.edu.au (S. Garic), ckondric@ucsd.edu (C. Kondric), ddn002@ucsd.edu (D. Nguyen).

0167-739X/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2004.09.010

Thus, on one hand one would like to have an easy possibility to test the importance of certain variables. On the other hand, computational parameterization of a biological system can lead to very useful models to better understand their behavior.

Our field of research is in computational chemistry and biochemistry. We computationally model the behavior of molecular systems, consisting of atoms with their associated nuclei and electrons, connected by chemical bonds. Biological macromolecules such as proteins, carbohydrates, DNA and RNA represent large and complicated assemblies of certain sets of smaller molecules or building blocks, and play important roles in life processes. However, small molecules also play important roles in our everyday life, for example, pharmaceuticals, hormones, energy transfer molecules, and so on. Of special interest in our studies are the interactions between all of these partners.

Due to their complexity, most (bio)molecular systems have many intrinsic parameters. For example, a large number of intra- and inte-rmolecular conformations and configurations are possible by changing the positions of the atoms within each molecule or between molecular partners. The study of these numerous conformations is an integral part of molecular dynamics and Monte Carlo simulations, and in molecular ligand-protein docking calculations. Additionally, one might also want to scan particular molecular properties within a large database of compounds, in order to understand some (bio)molecular phenomenon.

The investigation of molecular systems at the computational level involves the solving of fundamental physical equations such as the Schroedinger equation in quantum mechanics and Newton's laws in classical mechanics. Due to system complexities, researchers often need to summarize complicated details of these equations by using special parameters to provide approximated, but much less resource-demanding computational models of the actual systems. This involves such constructs as basis sets, pseudopotentials, and semiempirical methods on the quantum chemical electronic-structure level, and force fields, united-atom approaches, and scoring functions on the classical atomic and molecular level.

Parameter sweeps and, to a more limited extent dependent on the type of optimization algorithm, parameter optimizations have the major advantage that they generate lists of repetitive, independent calculations. This makes these kinds of studies embarrassingly parallel. Only at the beginning and end of each instance of a job, network transfer usually of input and output files from and to a central client machine is necessary. However, due to the complexity of the problem setup often a large number of compute-intensive calculations are necessary. Thus, a high throughput of jobs is desired, much more than can normally be executed on a single cluster or supercomputer. As such, these types of applications are perfectly suited for grid computing, i.e., the distribution onto a loosely internet-connected collection of compute resources.

In the past few years, grids in general have expanded the scope of what is possible in computational science regarding storage, computation, networking, and organization to new heights. However, it is very difficult for non-computer scientists to work directly with the grid middleware, such as the Globus toolkit [1]. Therefore, flexible software to build a bridge between the middleware and the user is necessary. The software is required to have an easy-to-use application program interface (API), and preferably a graphical user interface (GUI), and should allow a fast setup of new parametric modeling experiments. Automation of the preparation, submission, control, and collection of individual jobs should also be possible, but at the same time still be adaptable to a wide variety of application codes. All these conditions are fulfilled by the Nimrod toolkit [2], which we apply in our studies here. We should also mention that there are products analogous to Nimrod, such as the AppLeS Parameter Sweep Template (APST) [3], which are used for similar studies.

In this paper, we discuss how Nimrod enables us to solve parametric problems in quantum chemistry and molecular modeling on computational grids. We next give background on the architecture, capabilities, and usage of Nimrod as well as an overview of the grid testbeds employed in our experiments. We describe our most complete collaborative grid computing study up to now, the parameterization of a so-called group difference potential (GDP). This is followed by some ongoing work in our group in this area, concluded with a summary of the advantages and difficulties of applying grid computing to computational (bio)chemistry from our expe-

riences, plus a short future outlook for grid computing.

2. Setup—Nimrod software

Nimrod is a specialized toolkit for distributed parametric modeling. The software is actively developed in the group of David Abramson at Monash University, Australia, since 1994. Directed originally at local workstation clusters, Nimrod now targets wide-area computational grids. The main purpose is to enable scientists to design and perform parametric experiments, such as parameter sweeps and optimizations, on a computational grid quickly and easily. This means that only limited programming efforts and only minimal knowledge concerning the details of the grid setup itself should be necessary. Many research problems that are implied here would not be feasible without a grid infrastructure, but would also not be practicable without a parametric modeling system such as Nimrod (for examples, see Sections 4–7).

To reach this goal, Nimrod includes a simple declarative language to express a parametric experiment. Users simply generate a small plan or scheduling file that specifies their problem setup. This is not more difficult than writing a shell script, something computational scientists are typically familiar with. The first section of the plan file defines the names, types and domains (e.g., ranges and step sizes) of the parameters. Parameters can be floating point or integer numbers or text. In the case of parameter sweeps, the space resulting from building the cross product is scanned. The next section lists all the tasks that need to be performed in each instance of the experiment. Tasks include copying between the root machine and the grid nodes, substituting parameter placeholders by actual values in input files, executing shell commands, and behavior when an error occurs. In the case of parameter optimizations, a final section with details of the optimization algorithm follows.

The second necessary input to Nimrod is the list of local and remote compute resources available to the user on the grid. These are first generally specified by their name, and queue if necessary, and then individually selected for each parametric experiment. Furthermore, the middleware layer and corresponding security information (e.g., the Globus proxy), the type of hardware, the maximum number of jobs, the cost for each CPU cycle and other resource details are assigned (in part automatically, e.g., from the Globus MDS) to each machine. Currently supported services include Fork, Globus, PBS, and Condor. Nimrod then provides a uniform interface for communication with each resource.

After an experiment is started as either a parameter sweep (G) or a parameter optimization (O), Nimrod continuously discovers idle resources and automatically distributes and monitors the jobs produced for each single parameter set on the computational grid. The scheduling can consider various constraints such as a soft real time deadline or the overall cost of an experiment, that is, one can run the tasks as quickly and/or as cheaply as possible. Nimrod also keeps track of failed runs and reallocates calculations to different machines if necessary.

Internally, the core of the Nimrod system consists of a three-level PostgreSQL database that stores all the details of the experiments and resources. The individual jobs are actually run by a software component called the "agent". Agents are themselves scheduled and copied onto the various resources. The number of agents and type of binary on each machine is extracted from the resource entries in the database, and the set of commands to execute jobs determined by the middleware service (e.g., the Globus GRAM interface). Once initiated, the agents contact the database and request calculations, which are then interpreted and run sequentially on the resource. As such, any one agent can run more than one job. This design helps to hide the latency involved in the scheduling and invocation of a computation on a remote resource.

There are currently a number of implementations of Nimrod: *Nimrod/G* is the research prototype for parameter scans on a computational grid as described here, and provides the general basis for the other Nimrod tools. *EnFuzion* represents a commercial solution, based on the original Nimrod version for clusters. *Nimrod/O* enables execution of an arbitrary computational model as a core of a non-linear optimization process. The user specifies the type and space of parameters in the model, the starting point(s), possibly which constrains have to be fulfilled, and which output variable shall be minimized or maximized. *Nimrod/O* currently employs a number of built-in or interfaced optimization methods, namely domain subdivision, simplex search,

simulated annealing, BFGS, evolutionary and genetic algorithms, and additional methods can be relatively easily implemented. Jobs for cost function evaluation are then automatically distributed on a computational grid or cluster pool via the *Nimrod/G* or *EnFuzion* dispatchers.

Apart from the command line APIs, there are basically two GUIs for Nimrod: *Active Sheets* provides a spreadsheet interface for Nimrod/G, EnFuzion, and NetSolve. This GUI allows one to perform parallel computations on the grid, managed by spreadsheet software on a local desktop computer. The *Nimrod Portal* represents a web browser interface for Nimrod/G and Nimrod/O. This GUI permits users to set up, submit, and control their parametric experiments as well as to manage their resources and certificates from anywhere in the world. There is no need to port Nimrod to their internet client.

The outputs of the parametric experiments are generally sent back to the users by Nimrod in their native format, mostly as ASCII files. These files can then be used for further analysis and visualization. For example, in our studies we employed OpenDX for parameter space and QMView [4] for molecular model visualization. There is currently an interactive subjective optimization (OI) interface of Nimrod in development, requiring on-line interventions of the user into the parameter optimization process.

All non-commercial parts of the Nimrod distributed parametric modeling package are available from http://www.csse.monash.edu.au/∼nimrod/nimrodg/. Installation procedures for Unix systems are described there in detail; the only prerequisites are installations of PostgreSQL and Python. We are also developing an automated implementation of the Nimrod toolkit into a roll distribution for the ROCKS high-performance Linux cluster management software.

3. Setup—grid testbeds

One of the major tasks in grid computing studies is to collect the necessary number of computing resources to make a distribution of the calculations reasonable. We are very fortunate to participate in the Pacific Rim Applications and Grid Middleware Assembly (PRAGMA) collaboration of institutions, which contributed to our grid infrastructure. In addition, a few machines are available for use based on local agreements. This has allowed us in our past experiments to construct large, worldwide grid testbeds, in particular for the fourth PRAGMA workshop (PRAGMA4) and the 2003 Supercomputing conference (SC2003), where most of the extended parameter sweeps for the parameterization of the group difference potential (GDP) [6] for biomolecular computations were performed.

As can be seen from the snapshots of job distributions onto machines during the execution of Nimrod experiments at PRAGMA4 and SC2003, Fig. 1, the utilized resources (workstations, clusters and supercomputers) spanned a range of countries, institutions, and administrative domains as well as hardware architectures, operating systems, and queue managers. We based our grid testbed on the Globus toolkit, which provides a uniform interface for resource management, data management, information services, and security via its GRAM, GridFTP, MDS, and GSI components. Since Nimrod needs only to be available on the portal client, the setup on each server machine typically consisted of the following steps: (1) installation and configuration of the Globus toolkit and a certificate authority (CA), if not already existing; (2) setup of the user account and Globus certificate(s) under which the experiments are performed; (3) deployment of the application software.

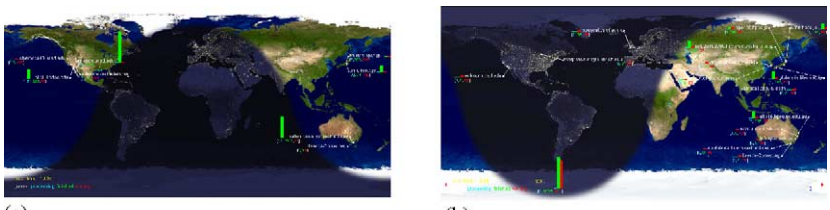


Fig. 1. Job distsributions during Nimrod experiments at (a) PRAGMA4 and (b) SC 2003.

Points (1) and (2) often provide difficulties due to differences in policy and system administrator experience between the involved sites. The participation of partners in a grid organization such as PRAGMA significantly eases these steps, due to agreed upon procedures. We also experienced several problems with such things as misconfigured Globus installations and minor Globus bugs. Therefore, the setup of the Globus toolkit on the resources is a critical issue for the success of such grid computing studies. Point (3) is of course dependent on the particular software package. For the GDP parameterizations, we used the quantum chemistry code GAMESS, [7,8] which is available for various hardware platforms and could thus be widely adopted. In practice, not all grid machines actually ran jobs, either due to problems in their software configuration or due to their workload. Fortunately, the highly dynamic nature of the grid is considered in the design of Nimrod, in that the decision about which machines are used on a particular experiment is deferred until execution time. Jobs can be automatically (re)scheduled, and network and node failures, common occurrences on a large computational grid, are adequately handled.

For our ongoing grid developments, we are in the process of establishing a smaller but more permanent grid testbed, based on the Globus and Nimrod packages. The *Resurgence Grid*, established between the University of Zurich and the San Diego Supercomputer Center (SDSC), includes a large four-processor Linux machine as main client and currently three Linux clusters as main servers; further inclusion of an IBM SP3 is planned.

4. Applications and examples

The Nimrod toolkit has been applied to a range of application areas, including bioinformatics, operations research, air pollution studies, laser physics, ecological modeling, discrete optimization and meta-heuristics, electronic CAD digital simulation, ad-hoc wireless network simulation design, public health policy, computational fluid dynamics, X-ray health standards, high gain antenna design, business process simulation, astrophysics and, as illustrated in this work, quantum chemistry. There is in principle no fundamental restriction to which types of parametric modeling problems the Nimrod package can be applied. However, for full benefit from distributing the calculations on a computational grid, the time spent in the computation part of the problem should outweigh the overhead induced by network transfer, I/O, queuing system scheduling, and the operations of Nimrod itself. Therefore, Nimrod is best suited for distributing many long-running, resource-demanding computations, as is typically the case for computational (bio)chemistry.

5. Parameterization of a group difference potential

Computational studies involving large and complex biological structures often entail numerical experimentation and hypothesis testing. When a parameter space is searched for optimal solutions, the corresponding computational requirements are amplified by several orders of magnitude. Theoretical modeling of extended molecular systems is challenging due to the rather large number of atoms, the vast conformational space which needs to be sampled dynamically, and the fine spatial and temporal resolution required, all needing sophisticated techniques to correctly describe the physical and chemical behavior.

Hybrid quantum mechanics–molecular mechanics (QM/MM) methods [9] are often used for such studies. In such techniques, the large molecular system is divided into two regions. A small, chemically “active” region is described by accurate approaches based on the Schroedinger equation (QM), while the surrounding larger, “inactive” region is treated with more approximate classical force field methods (MM). Unfortunately, the physical concepts behind the two techniques are so different that they cannot be easily coupled. In particular, when chemical bonds are cut between both parts, dangling bonds in the MM region can simply be eliminated. The outermost atoms of the QM region however, which involve the detailed nature of the electrons, would become radical in nature and behave completely different than required if the bonds were simply removed. One way to accommodate such atoms includes the first atom of the MM part as a capping atom in the QM computation, parameterized such that it correctly reflects the properties of the cut bond. This method does not require extended changes in the source code and does not lead to problems with the handling of artificial link atoms.

Zhang et al. recently developed such a "pseudobond" approach for sophisticated QM-based methods [10]. They added an effective core potential (ECP) to a fluorine atom to model the methyl group in a carbon–carbon single bond. Their parameterizations were done for the ethane molecule, CH_3CH_3, which becomes CH_3C_{ps}, (C_{ps} = F with pseudopotential), then tested on ethane derivatives, and later applied to investigate several enzymatic reactions using QM/MM techniques. Unfortunately, we found serious instabilities in the Zhang et al. pseudobond parameterization. Furthermore, pseudoatom ECPs appear rather difficult to parameterize due to the diversity of their target features, the multi-dimensionality of function and parameter space, and the occurrence of local minima in the optimization procedure.

Therefore, we have developed a new effective pseudoatom potential, [5,6] which only deals with the discrepancies between the isoelectronic CH_3 and F groups without exchanging the core. It is thus named "group difference potential" (GDP). The potential also provides direct information about the substituent effects of functional groups, which play a large role in applications from organic chemistry to pharmaceutical drug development. In addition, the potential may be gradually switched on or off, potentially facilitating QM free energy difference determinations. A superposition of two unmodified Gaussian functions turns out to be the most appropriate functional, a format which is already implemented in many QM programs:

$$U_{\text{eff}}(r) = A_1 \exp(-B_1 r^2) + A_2 \exp(-B_2 r^2)$$

This form corrects for differences in electron interaction and basis set through negative (attractive) and positive (repulsive) values for the amplitude coefficients A_1 and A_2. The positive exponential prefactors B_1 and B_2 specify the radial extent of each term around the fluorine atom; the smaller their value, the less compact the corresponding function, and vice versa.

To analyze the results for each parameter set, a suitable cost function that reduces the differences between the properties of ethane and pseudoethane to a single number is required. Bond lengths, bond angle, dissociation energy, Mulliken overlap populations and atomic charges were selected as independent properties to optimize with respect to. We applied a common level of QM theory (B3LYP/6-31G(d)) using both spherical and Cartesian basis set formats. This results in four calculations for every A_1, A_2, B_1, B_2 set, two on pseudoethane and two on the corresponding pseudomethyl radical. All computations were done with the GAMESS quantum chemical program package. The resulting 32 properties identified (hydrogen values all appear three times) have diverse units, accuracy, and importance. As such, the differences between actual and target properties x_i and X_i must be weighted appropriately for the overall cost function by applying a normalized least squares expression

$$f(A_1A_2B_1B_2) = \frac{1}{\sum_{i=1}^{32} w_i} \sum_{i=1}^{32} w_i \left(\frac{x_i - X_i}{u_i} \right)^2$$

The "weighting" factors w_i correct for the number of occurrences of each carbon or hydrogen feature. The "unifying" factors u_i reflect the apparent, or desired, accuracy of each property and are chosen from chemical intuition. This equation is evaluated after completion of each tuple of GAMESS jobs, so that the w_i and u_i values can be easily adjusted to later allow straightforward pseudopotential optimizations.

To identify interesting low-cost regions and avoid trapping in local minima, we first scanned portions of the parameter space in their entirety. This task consists of huge numbers of short, uncoupled QM calculations and thus quite suited for grid computing. In the initial experiment at the PRAGMA4 workshop, the four variables were varied between −10 and 10 a.u. in steps of 1 a.u. for A_1 and A_2, and between 0 and 10 a.u. in steps of 2 a.u. for B_1 and B_2. Ignoring function symmetry for now, this leads to $15{,}876 \times 4$ jobs. At the SC'2003 event, we performed even larger parameter sweeps with $53{,}361 \times 4$ and $60{,}016 \times 4$ individual calculations.

A snapshot from the initial parameter scan at PRAGMA4 is displayed in Fig. 2. This snapshot shows the complexity of the cost function hypersurface. The isosurfaces reveal a system of connected channels with a rather "bumpy" terrain.

To find appropriate starting points for subsequent optimizations, regions with function values lower than the "zero" model (no GDP or canceling GDP red isosurface in Fig. 2) are of special interest. Several such "local" minima are scattered over the parameter space with no apparent pattern, although further analysis suggests partial linear dependence. Although we were able to find a 'minimum' in this particular run, all cost function

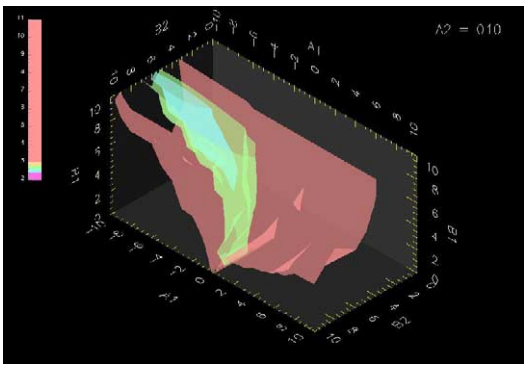


Fig. 2. A single snapshot of the GDP parameter space scanned in the PRAGMA4 experiment. A_1, B_1, and B_2 are displayed on the *x*, *y*, and *z* axes; A_2 evolves with the snapshots. Isosurfaces of the cost function *f* are drawn at 3620 (*red*), 631 (*green*), 398 (*cyan*), and 251 (*magenta*).

values were still too high to be acceptable. Therefore, in the SC2003 experiments we also performed sweeps with a logarithmic point distribution and a higher grid density around the most interesting region. The deepest minimum appears when a medium-size repulsive and a diffuse attractive Gaussian function are combined to build a maximum potential at the fluorine core and a shallow depression closer to the bond distance of the neighboring carbon atom. We tentatively attribute this to the larger size and smaller electron attraction of a methyl group compared to a fluoro group. With these very large data collections in hand, it turned out that the significance of the best "global" minimum could be remarkably improved when the unit factor for the bond angle was reduced, a fact that can not be easily deduced from chemical intuition. This revealed the most promising parameter region and weighting for subsequent GDP optimizations with Nimrod/O.

In subsequent studies, we used the resulting optimal parameter combinations as starting points for additional grid runs. The Nimrod/O tool, which performs automatic optimization was incorporated to explore the parameter space more intelligently with a number of search heuristics, to find the final optimized GDP. As explained above, we varied variables in the least squares procedure to generate a more funnel-like hypersurface. This will allow minimizations commencing from any remote place to travel towards the global minimum more directly. Overall, this procedure considerably reduces the development time of GDPs for use in studies of many types of molecular systems and reaction processes using the GAMESS software.

6. Protein–ligand docking studies

In a new project in the field of molecular modeling, we have begun to apply Nimrod/G and Nimrod/O to investigations in protein–ligand docking. Such types of studies are of high importance for drug design in the pharmaceutical industry. The goal here is to scan possible conformations for combined complexes of ligand (often a small organic molecule) and protein (for example an enzyme that needs to be blocked) in space, and to find their optimal position(s) with respect to each other. The tendency for them to bind together is described by a free energy cost function. In the current setup, we use Nimrod to scan through the possible conformations of a test system based on a rotation and translation algorithm, and distribute the corresponding jobs onto a grid of machines. Unreasonable structural configurations resulting when atoms are either too far apart, or too near each other, are excluded. The electrostatic part of the binding free energy is then calculated using an adaptive Poisson–Boltzmann solver, in this case the APBS software [11]. An additional goal in this work is to refine the scoring energy function for this procedure.

7. Workflow and database investigations

The pseudopotential and docking examples demonstrate the suitability of Nimrod for high-throughput calculations in computational (bio)chemistry. Another interesting application is therefore to utilize the procedure to perform calculations for a large set of molecules, putting the results directly into a database. One of the main problems of such an endeavor is that multiple complicated steps are necessary both to prepare the input files for the calculations, as well as to analyze and store the output files into a relational database. This goes beyond the scope of the short pipelines possible to define in Nimrod plan files. However, such preparation and storage steps are not very compute-intensive, so they can be run on a local host. Therefore, we use Nimrod/G just to distribute the resource-demanding

parts of the calculations on a computational grid on basis of simple plan files, which are automatically generated from templates. The control of Nimrod/G itself is embedded in a more complex workflow infrastructure with GUI, based on the Ptolemy II [12] and Kepler [13] systems. The current setup targets the automatic calculation of jobs with GAMESS. However, the overall design is very general such that adaptation to a number of different scientific computation codes should be quite possible.

8. Conclusions

We have illustrated how the Nimrod toolkit has made it possible for us to solve parametric modeling tasks for molecular modeling on computational grids. We were able to parameterize a GDP pseudopotential [5,6] which can be used now for applications in QM/MM or free energy difference calculations, which otherwise would have been very difficult. This has encouraged us to expand our studies into the area of protein–ligand docking. Furthermore, we are currently building a workflow infrastructure on top of Nimrod, which will permit seamless generation, distribution, computation, storage, and analysis of computational (bio)chemistry jobs in an automated, high-throughput manner [14]. In this way, we hope to enable grid computing as an every-day tool for researchers interested in performing molecular modeling studies.

For scientists in general, the Nimrod toolkit simplifies the execution of parameter sweeps and optimizations on a computational grid and thus significantly enhances the usability of the global grid for such studies. Most importantly, it hides the underlying grid middleware and makes the management of the resources and the automation of the jobs user-friendly. However, it cannot and is not meant to solve all the issues associated with setting up a computational grid, such as policy differences between the different involved sites as well as installation and configuration of the grid middleware. In addition, while Nimrod considerably eases the use of grids for repetitive high-throughput calculations with existing scientific software, it is not capable of distributing high-performance calculations with intrinsically grid-parallelized codes. Yet, the diverse hardware, loose network connections, uncontrolled node failures, and different resource management favors the use of computational grids for parameterization applications.

Acknowledgements

We thank PRAGMA and all organizations that provided computing resources, and their system administrators, in particular the SDSC ROCKs group. We thank J.P. Greenberg and K. Thompson, SDSC, for software installations. K.B. acknowledges NSF-DBI-0078296, ANI-0223043 and NIH-NBCR-RR08605. We thank D. Kurniawan at Monash for visualizations. The Nimrod project is supported by DSTC and GrangeNet, funded in part by Australian Government. D.A. achknowledges the APAC. C.K. and D.N. thank the PRAGMA-PRIME project for funding summer internships at Monash for work described here.

References

[1] I. Foster, C. Kesselman, Intl. J. Supercomput. Appl. 11 (1997) 115.
[2] Nimrod, 2004. http://www.csse.monash.edu.au/~davida/nimrod.
[3] H. Casanova, G. Obertelli, F. Berman, R. Wolski, Proceedings of the Supercomputing Conference (SC'2000), 2000.
[4] K.K. Baldridge, J.P. Greenberg, J. Mol. Graph. 13 (1995) 63.
[5] W. Sudholt, K.K. Baldridge, D. Abramson, C. Enticott, S. Garic, New Generat. Comput. 22 (2004) 137.
[6] W. Sudholt, K.K. Baldridge, D. Abramson, C. Enticott, S. Garic, Lect. Notes in Comput. Sci. 3036 (2004) 6748.
[7] M.W. Schmidt, K.K. Baldridge, J.A. Boatz, J.H. Jensen, S. Koseki, M.S. Gordon, K.A. Nguyen, T.L. Windus, S.T. Albert, Quantum Chem. Program Exch. Bull. 10 (1990) xx.
[8] M. Schmidt, K.K. Baldridge, J.A. Boatz, S. Elbert, M. Gordon, J.H. Jenson, S. Koeski, N. Matsunaga, K.A. Nguyen, S.J. Su, T.L. Windus, M. Dupuis, J.A.J. Montgomery, Comp. Chem. 14 (1993) 1347.
[9] J. Gao, Acc. Chem. Res. 29 (1996) 298.
[10] Y. Zhang, T.-S. Lee, W.T.J. Yang, Chem. Phys. 110 (1999) 46.
[11] N.A. Baker, D. Sept, S. Joseph, J.A. McCammon, Proc. Natl. Acad. Sci. USA 08 (2001) 10037.
[12] Ptolemy II, 2004, http://ptolemy.eecs.berkeley.edu/ptolemyII/.
[13] http://Kepler.ecoinformatics.org/, http://www.Kepler-project.org/.
[14] K.K. Baldridge, J. Greenberg, W. Sudholt, I. Altintas, K. Bhatia, S. Mock, C. Amoreira, Y. Potier, IEEE on Grid Comput. 2004.



Wibke Sudholt is a postdoctoral fellow with Kim K. Baldridge at the University of Zürich, Switzerland, and also recently joined the CoLab under Petros Koumoutsakos at ETH Zürich half time. In 2001, she obtained her doctoral degree (Dr. rer. nat.) with Wolfgang Domcke at the Heinrich-Heine-University Duesseldorf, Germany, for theoretical studies of a charge-transfer process. She then was a DAAD postdoctoral fellow with J. Andrew McCammon and Kim K. Baldridge at the University of California, San Diego. Her current interests include hybrid methodologies combined with grid computing infrastructure and scientific workflow environments for computational chemistry applications.



Kim K. Baldridge is a theoretical and computational chemist with expertise in the design, development, implementation, and application of computational methodology, for understanding chemical and biochemical reaction processes. Efforts include development of computational tools and associated grid technologies for the broader community. She has over 100 publications and is a co-author on the widely used computational chemistry software, GAMESS. Baldridge is a Fellow of the AAAS and the APS, and holds the 2000 Agnes Fay Morgan Award. She is currently a Professor of Theoretical Chemistry and director for grid applications at the University of Zürich, and a Distinguished Scientist at the San Diego Supercomputer Center, California.



David Abramson has been involved in computer architecture and high performance computing research since 1979. Abramson is currently a Professor of Computer Science in the School of Computer Science and Software Engineering (CSSE) at Monash University, Australia. Prior to joining Monash University in 1997, he has held appointments at Griffith University, CSIRO, and RMIT. He is a project leader in the Co-operative Research Centre for Distributed Systems Nimrod Project and also Chief Investigator on two ARC funded research projects (Guard and GriddLeS). Abramson has chaired a number of international conferences. He has published over 100 papers and technical documents.



Colin Enticott completed B Comp (Hons) July, 2002 at Monash University. His project, which he did under the supervision of Professor David Abramson, "The Multi Site EnFuzion Client" dealt in the area of cluster -of-clusters computing that has lead him into Grid computing. Hired by Distributed Systems Technology Centre, the majority of his work is with assisting scientists using their applications with Nimrod and developing the user front-end of Nimrod, the Nimrod Portal.



Slavisa Garic completed a Bachelor of computer Science (Hons) degree at Monash University, Australia in November 2001. His project, Suburban Area Networks: Security, involved working on security aspects of wireless community/suburban networks. At the start of year 2002 he joined Distributed Systems Technology Centre, Melbourne Australia, where he currently works as a core Nimrod/G developer.

Chris Kondric is currently an undergraduate student at the University of California, San Deigo. Over the summer of 2004, he spent 6 weeks at the University of Monash under the PRAGMA-Prime program, working with the Nimrod team on the development of computational grid tools for molecular docking studies.

Duy Nguyen is currently an undergraduate student at the University of California, San Deigo. Over the summer of 2004, he spent 6 weeks at the Monash University under the PRAGMA-Prime program, working with the Nimrod team on the development of computational cluster application software, in particular the Nimrod ROCKS roll toolkit.

An Atmospheric Sciences Workflow and its Implementation with Web Services

David Abramson §, Jagan Kommineni, §, John L. McGregor ‡ and Jack Katzfey ‡

§ School of Computer Science and Software Eng., Monash University, 900 Dandenong Rd, Caulfield East, 3145, Australia

‡ Division of Atmospheric Science, CSIRO, PMB 1, Aspendale,Vic, 3195, Australia

Abstract: Computational and data Grids couple geographically distributed resources such as high performance computers, workstations, clusters, and scientific instruments. Grid Workflows consist of a number of components, including: computational models, distributed files, scientific instruments and special hardware platforms. In this paper, we describe an interesting grid workflow in atmospheric sciences and show how it can be implemented using Web Services. An interesting attribute of our implementation technique is that the application codes can be adapted to work on the Grid without source modification.

1. Introduction

Computational and data Grids couple geographically distributed resources such as high performance computers, workstations, clusters, and scientific instruments. Accordingly, they have been proposed as the next generation computing platform for solving large-scale problems in science, engineering, and commerce [4][5]. Unlike traditional high performance computing systems, such Grids provide more than just computing power, because they address issues of wide area networking, wide area scheduling and resource discovery in ways that allow many resources to be assembled on demand to solve large problems.

Grid applications have the potential to allow real time processing of data streams from scientific instruments such as particle accelerators and telescopes in ways which are much more flexible and powerful than are currently available. Of particular interest are applications, called “Grid Workflows”, that consist of a number of components, including: computational models, distributed files, scientific instruments and special hardware platforms (such as visualisation systems) [3]. Importantly, such workflows are interconnected in a flexible and dynamic way to give the appearance of a single application that has access to a wide range of data, running on a single platform. Grid workflows have been specified for a number of different scientific domains including physics [6] and gravitational wave physics [2].

In this paper we describe a Grid workflow for solving problems in atmospheric sciences. The workflow supports the coupling of a number of pre-existing legacy computational models across distributed computers. An important aspect of the work

is that we do not require source modification of the codes. In fact, we don't even require access to the source code. In order to implement the workflow we overload the normal file IO operations to allow them to work in the Grid. We also leverage existing Grid middleware layers like Globus [4] [5] to provide access to control of the underlying resources.

In Section 2 we describe an atmospheric science workflow, with some detail of the functions of the various components. Section 3 discusses the implementation techniques, and Section 4 provides some experimental results.

2. An Atmospheric Sciences Workflow

Global climate models: A global climate model is a computer model representing the atmosphere, oceans, land and sea-ice. By solving mathematical equations based upon the laws of physics, a GCM simulates the behaviour of the climate system. The model divides the planet into a number of vertical layers representing levels in the atmosphere and depths in the oceans, and divides the surface of the planet into a grid of horizontal boxes separated by lines which may be similar to latitudes and longitudes. In this way, the planet is covered by a three-dimensional grid of boxes (Figure 1).

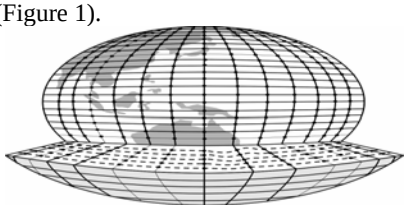


Fig 1 - Representation of the Earth's surface and atmosphere in a typical global climate model.

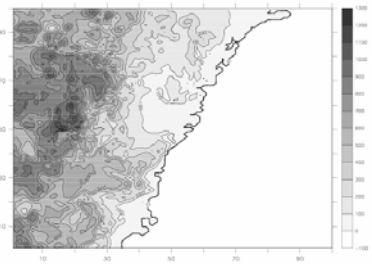


Fig 2 - DARLAM model domain for simulations using a 3 km grid over Sydney.

Global climate models capture large scale features like the deserts and tropics very well, but have difficulty capturing smaller features like cyclones and thunderstorms because they occur at scales much smaller than the grid boxes.

Regional climate models: To improve regional detail in climate models, it is desirable to reduce the spacing between grid points. However, due to the complexity of global climate modeling, computational requirements become prohibitive if the horizontal grid resolution is less than a few hundred kilometers. At this resolution, vitally important small-scale phenomena, like tropical cyclones and cold fronts, are poorly captured. This affects simulated patterns of temperature and rainfall, and hence the ability to realistically simulate observed regional climate features in GCMs.

A computationally feasible alternative to a coarse resolution global climate model is to use a finer resolution model over a small part of the globe. A regional climate model (RCM), with a horizontal resolution of about 100 km or less, is able to simulate regional weather patterns better than most GCMs [8][11][7]. Part of the reason for the

improved climate simulation relative to GCMs is the fact that coastlines and mountains are represented in more detail in RCMs. Since topographic features strongly influence regional temperature and rainfall, more detailed features are likely to give a better climate simulation.

A regional climate model requires meteorological information at its lateral boundaries in order to simulate weather within its boundaries. For climate change studies, an RCM is typically driven at its boundaries by information from a coarser-scale GCM. This is commonly called nesting an RCM inside a GCM. One-way nesting allows information to flow from the GCM to the RCM each simulated day, but the weather simulated by the RCM does not affect the GCM interactively. This means that the RCM can be run after the GCM experiment has been completed.

A Grid Workflow: Traditionally, the GCM and the RCM have been executed on the same computer system, and data is passed between them using conventional files. However, the computational Grid discussed in Section 1 provides an ideal framework for executing these models on different machines that are physically distributed. There are many reasons why one might wish to do this. First, both models may have different computational requirements and be suited to different types of platforms. For example, one may execute well on a vector supercomputer and the other may be efficient on a parallel processor. Second, both models may not have been ported to the same hardware. Thus, it may be time consuming and expensive to couple them on one machine. Third, the models may be "owned" by different organizations. As discussed in [4] the computational grid facilitates the construction of a "virtual organization" in which the models are linked into a single grid application without actually moving the codes to a single organization. Finally, it may be possible to pipeline the computations, providing a quicker solution than if they were run sequentially on one system. This can be achieved if the data files are replaced by communication pipes that allow one program to write data concurrently with a downstream one reading the same data. Figure 3 shows such a grid workflow based around the models discussed above. In this paper we discuss a particular system involving three models – a GCM called C-CAM, a RCM called DARLAM and a data filter called cc2lam.

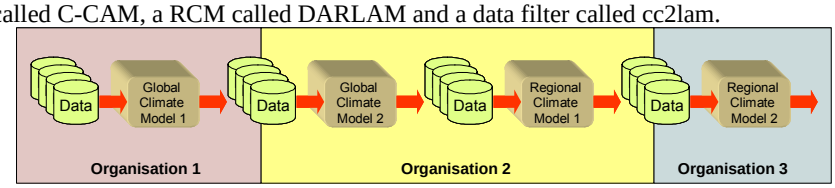


Fig 3 - An Atmospheric Sciences Grid Workflow

3. An Implementation with Web Services

Clearly it is possible to implement the system described in the previous section in a number of ways. For example, the programs could be run unmodified, and some system could be responsible for copying files from one machine to another. This is effectively the practice that atmospheric scientists have employed manually for some time now. Or, the files could even be shared by a single distributed file system like NFS or AFS, removing the need to explicitly copy them from one local file system to

another. The major disadvantage of this general approach is that it does not take advantage of any potential overlap in the computations. Another option is to modify the programs so that they do not read directly from files, but instead they use a message passing library like PVM or MPI to send data from one model to another. This allows the computations to be pipelined, however the programs would need to be modified at the source level. Further, once modified that would no longer work as stand alone codes, limiting the flexibility of the individual components. A third alternative is to modify the file system library so it performs message passing rather than writing to and reading from local files. This achieves the advantages of both of the previous alternatives, requires no source modification (only relinking the application) and does not permanently fix the way the programs operate. For example, by linking the normal file system primitives, the programs behaves as normal, reading and writing local files. However, when the message passing library is linked, the program sends messages for writes, and receives messages for read. In a previous paper we proposed such a mechanism, called NetFiles, for implementing parallel master-slave programs [1]. NetFiles were implemented within a single cluster or parallel machine and could not cross administrative domains. Here we have broadened the approach to support interprocess communication across the computational Grid. Accordingly we have called this mechanism GridFiles. Figure 4 shows how two legacy computations can be coupled using GridFiles.

In the GridFiles approach, the conventional system calls (like open, read, write etc.,) are replaced (transparently) by a call to a module called a “File Multiplexer”. The File Multiplexer is responsible for passing the file system operations onto an appropriate service, and has the flexibility to change the mappings dynamically. Thus, the File Multiplexer can redirect IO requests to local files, local processes, remote files or remote processes. This means that a program can perform a READ operation, and this might read a local file, or a remote one, or even be connected directly to a WRITE operation on a remote machine. The latter mode is the equivalent of connecting the two programs by sockets, and allows complete overlap of IO operations, and is called the buffer service. This structure is depicted in Figure 4. The File Multiplexer is composed of three clients. The Local File Client performs local file operations. The Grid File Client communicates with the Grid Buffer Service, and the GNS Client communicates with the GriddLeS Name Service. The Grid Buffer and Name Services are both implemented using Web Service technologies such a SOAP and XML [9][10].

The GridBuffer service acts as a sink for WRITE operations and a source for READs. In order to support random read and write operations, data is stored in a hash table rather than a sequential buffer. Thus, if a read is issued for a block that has not been written yet, the read waits for the data to arrive. After a block has been read from the hash table it is written to a cache file and then deleted from the hash table. The cache file is provided for two main reasons. First, it allows a block to be reread even after it has been deleted from the hash table. This occurs when the reader seeks back to a previous block. Second, it provides a mechanism for implementing broadcast operations to more than one process. When this happens, the first reader obtains the data from the hash table, but subsequent readers retrieve the data from the cache file. It is possible to have an arbitrary number of readers using the approach without the need to inform the Grid Buffer Service before hand.

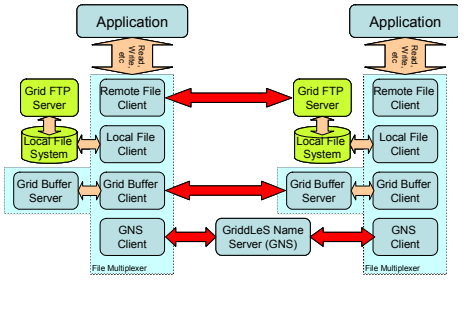


Figure 4 - Using GridFiles to link apps

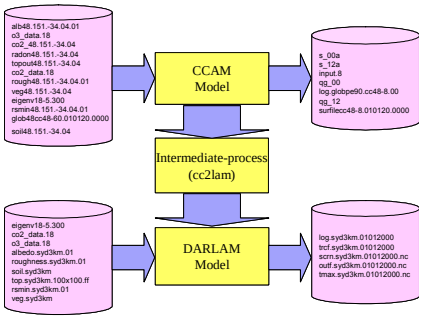


Figure 5 – File dataflow

The GriddLes Name Service (GNS) is responsible for configuring the grid application. Each entry in the GNS indicates what should happen when a particular file is opened on a particular resource. At the time of the deployment the GNS reads the data from the configuration file and keeps it in a hash table. These keys and values correspond to the global naming schema. The reader and writer applications can use different keys which are mapped to the same entry in the GNS, indicating both are linked to either the same file or buffer. Buffers are distinguished from files by the word "buffer" in the value. If an entry in the GNS represents a buffer, then additional configuration information is contained. The GNS can be updated at any time, a this reconfigures the Grid application dynamically.

All the applications and web services can run on the same system or each one can be distributed to the different Grid nodes which are located geographically at different locations.

4. Experimental Results

Figure 5 shows a particular configuration of the three atmospheric models discussed in section 2, namely C-CAM, cc2lam and DARLAM. Importantly, most of the computation is performed by C-CAM and DARLAM, and cc2lam provides simple data manipulation and filtering between the two codes. In this section we describe an experiment in which C-CAM, cc2lam and DARLAM are coupled by Grid Buffers, that is, the output of C-CAM is streamed into DARLAM (via cc2lam). Because we used a file multiplexer as discussed in section 3, *it was possible to do this without source modification.* This is important because C-CAM and DARLAM are legacy codes written in Fortran, and we did not wish to make significant modifications to their structure. This scheme is very flexible. All models and services can run on a single machine, or on different nodes of a cluster or on different machines in a computational grid. Further, these configuration changes can occur without any modification to the atmospheric models.

When the first writer application (C-CAM) writes block of data (typically of 4096 bytes) using the write statement, then the data need to be transferred over to the GridBuffer service by using the client component of GridBuffer service which is in FMP underneath the application layer without the user interaction. Once the data arrives, the reader application (in this context cc2lam) uses the GriddBuffer client and

obtain the block of data and writes into the other GridBuffer in a similar way, so that the other reader application (in this context the DARLAM Model) reads the block of data. In some instances, DARLAM rereads some of the input data. Because the data has already been deleted from the hash table in the Grid buffer Service, it is read form the cache file instead. This occurs transparently to the DARLAM model.

In this experiment we utilised a number of machines shown in Table 1, in three countries (AU, US and UK).

Table 1 Machine List

Name	Address & Details	Country	Name	Address & Details	Country
dione	dione.csse.monash.edu.au Pentium 4, 1500 MHz, 256 MB, Redhat Linux 7.3	AU	brecca	brecca-2.vpac.org Intel Xeon, 2.8 GHz, 2048MB, Redhat Linux 7.3	AU
freak	freak.ucsd.edu Athlon Processor, 700 MHz, 256 MB, i386, Debian	US	bouscat	bouscat.cs.cf.ac.uk Pentium 3, 1 GHz, 1544 MB, Red Hat Linux 7.2	UK
vpac27	vpac27.vpac.org Pentium 3, 997 MHz, 256 MB, Red Hat Linux 7.3	AU	dragon	dragon.vpac.org Intel(R) Pentium(R) 4 CPU 1.70GHz, 1006MB Red Hat Linux 8.0 3.2-7	AU

In this case study, three different experiments were performed.

Case 1: All models are executed concurrently on the same machine. There are two variations on this experiment. First, the programs read and write conventional local files. The results of this experiment are shown in the "Files" column in Table 3. Second, the programs use Grid Buffers instead of files. These times are shown in the Buffers column in Table 3. All times are cumulative, and thus the DARLAM time also indicates the total time taken. In this experiment the code was run for 480 and 960 time steps.

Table 2 – Cumulative concurrent runs on the same system (time in hr:min:sec)

		480 time steps		960 time steps	
Computer	Model	Files	Buffers	Files	Buffers
dione	C-CAM	00:41:18	00:44:10	01:23:57	01:29:59
	cc2lam	00:41:56	00:44:15	01:25:13	01:30:09
	DARLAM	01:08:17	00:49:12	02:19:13	01:35:00
brecca	C-CAM	00:18:13	00:20:05	00:36:29	00:41:21
	cc2lam	00:18:25	00:20:12	00:36:50	00:41:24
	DARLAM	00:27:58	00:22:57	00:56:35	00:44:11
freak	C-CAM	00:34:35	00:35:21	01:22:28	01:28:15
	cc2lam	00:35:26	00:35:33	01:23:15	01:30:01
	DARLAM	00:52:39	00:40:30	02:14:22	01:36:15
bouscat	C-CAM	01:10:22	01:17:51	02:44:03	02:42:42
	cc2lam	01:10:39	01:18:10	02:44:39	02:43:00
	DARLAM	01:55:27	01:29:59	04:14:54	02:54:55
dragon	C-CAM	00:41:25	00:41:28	01:23:01	01:27:57
	cc2lam	00:41:46	00:41:41	01:23:43	01:28:08
	DARLAM	01:06:17	00:46:24	02:13:12	01:32:57

The results shown in Table 2 highlight that using buffers is always faster than using files when the codes are run on the same system. This is interesting because the models are multiprocessing the single CPU on these machines, and thus it suggests

that buffers are allowing some overlap of computation and communication. Even more interesting is that most of the runs performed with buffers were actually faster than running the codes sequentially on the same platform, again because of the overlap in IO and computation. The exceptions to this are those run on dione and vpac27, which are presumably because of the relative speed of the computation and the IO on these two machines.

Case 2: C-CAM and DARLAM are executed on different computers at different locations, whilst cc2lam is run on the same machine as C-CAM. Because there are a large number of potential pairings of machines, we have selected a few interesting ones and present the timing results in Table 3. Again all times are cumulative, and thus the DARLAM time also indicates the total time taken. In the case where local files are written we have also included the time taken to copy the files in the cumulative totals. The runs were done for both 480 and 960 time steps.

Table 3 – Cumulative concurrent runs (Time in hr:min:sec)

		Close Systems					Distant Systems			
		480 Time steps		960 Time steps			480 Time steps		960 Time steps	
Model	Mach	Files	Buffers	Files	Buffers	Mach	Files	Buffer	Files	Buffer
C-CAM	dione	00:28:21	00:34:20	00:55:18	01:08:14	brecca	00:16:34	00:18:39	00:32:32	00:38:14
cc2lam	dione	00:28:29	00:34:32	00:55:40	01:08:25	brecca	00:16:42	01:00:48	00:32:49	01:48:42
File Copy		00:29:19		00:56:52			00:24:12		00:40:25	
DARLAM	vpac27	01:00:29	00:48:47	02:00:59	01:26:04		00:56:04	01:05:03	01:46:19	01:52:58
						bouscat				
C-CAM	brecca	00:16:34	00:18:37	00:32:32	00:38:55		01:07:29	01:18:18	02:29:58	02:22:44
cc2lam	brecca	00:16:42	00:18:44	00:32:51	00:39:01	bouscat	01:07:41	01:30:34	02:30:23	02:34:12
File Copy		00:16:57		00:33:04			01:17:11		02:37:59	
DARLAM	vpac27	00:47:57	00:40:43	01:37:11	01:14:35	bouscat	01:24:57	01:31:55	02:55:02	02:35:20
						brecca				
C-CAM	brecca	00:16:34	00:18:05	00:32:32	00:38:55	brecca	00:16:34	00:18:20	00:32:32	00:37:10
cc2lam	brecca	00:16:42	00:18:12	00:32:48	00:39:01	brecca	00:16:42	00:33:49	00:32:49	01:05:40
File Copy		00:17:32		00:33:59			00:20:17		00:37:14	
DARLAM	dione	00:30:48	00:25:10	00:59:53	00:44:27	freak	00:33:55	00:41:45	01:07:01	01:07:19
C-CAM	brecca	00:16:34	00:18:37	00:32:32	00:39:03	freak	00:30:31	00:36:57	01:16:56	01:14:03
cc2lam	brecca	00:16:42	00:18:40	00:32:48	00:39:18	freak	00:31:01	00:44:08	01:17:23	01:21:07
File Copy		00:17:32		00:33:04			00:34:36		01:21:48	
DARLAM	dragon	00:29:44	00:23:10	01:00:09	00:43:53	brecca	00:42:22	00:45:17	01:38:51	01:22:16

The results show that buffers are always faster for systems which have good network connections than when files are used. Interestingly, the results for 960 time steps are less than double those for 480. This is because the startup overheads are masked in the longer runs, and thus the parallel efficiency is higher.

The results for distant machines tell a different story. Because these machines have poorer networks between them, it is not always faster to use buffers than to copy the files. For example for the shorter runs of 480 time steps, it is always better to use file copies. On the other hand, buffers are more efficient for some of the longer 960 time step runs. The results shown here highlight the importance of being able to reconfigure the application dynamically because it is not always possible to know which configuration will be more efficient.

Case 3: In this experiment C-CAM and cc2lam are executed on brecca and DARLAM model is run on dragon. We run models for 480 time but data is exchanged at different intervals from 60 to 15 time steps. The results are shown in Table 4.

Table 4 – Varying the output frequency (Time specified in hr:min:sec format)

	Amount of Output Data produced by different models in MB				
Print Interval in time steps	CCAM	cc2lam	DARLAM	Total Time with Files	Total time with Buffers
60	72.663	42.485	43.000	00:29:38	00:23:23
30	137.253	80.249	83.599	00:35:50	00:26:30
15	264.428	155.771	163.760	00:38:54	00:26:21

As expected, as the write interval reduces, the magnitude of data produced by each model increases. Even though there is an increase in computation time as the write interval reduces, the total computation time with the buffers is less than with files because more overlap is possible.

Table 5 analyses the degree of overlap in the computations, and therefore the efficiency, for a few different machine configurations. Here we calculate the best theoretical time that could have been achieved assuming no startup costs and perfect networking, and compare this to the actual time. The results indicate that the efficiency can be quite high for low latency, high bandwidth networks, especially for the longer runs.

Table 5 Overlap comparisons (Time specified in hr:min:sec format)

Machines	Time Steps	C-CAM with Files	DARLAM with Files	Total time with Buffers	Best Theoretical	Efficiency
Brecca-2 & vpac27	480	0:16:34	0:31:00	0:40:43	0:31:00	76%
	960	0:32:32	1:04:07	1:14:35	1:04:07	86%
dione & vpac27	480	0:28:21	0:31:00	0:48:47	0:31:00	64%
	960	0:55:18	1:04:07	1:26:04	1:04:07	74%
Brecca-2 & vpac27	480	0:16:34	0:13:16	0:24:58	0:16:34	66%
	960	0:32:32	0:25:54	0:44:27	0:32:32	73%

5. Conclusions

In this paper we have discussed the implementation of a grid workflow that couples two legacy atmospheric science applications, namely a global climate model and a regional weather model. *One of the more significant achievements of the experiment is that we managed to do this without any changes in the source code of the two models.* This is no mean feat since they are legacy codes written in Fortran and were designed without any knowledge of the underlying grid infrastructure. Inter-process communication is provided by a software device called a File Multiplexer, and we have chosen to implement this with web services.

The performance results indicate that there are cases when it is advantageous to couple the models tightly using pipes, and other cases where it is more efficient to write files locally, copy them to the receiving node and read them locally. An important feature of our implementation is that the decision about whether to copy or use buffers can be delayed until *the time that the application is configured* and does

not need to be integrated into the source code of the models. We are not aware of other systems with equivalent flexibility.

The case study discussed here is actually a small fragment of a much larger system currently being constructed. Rather than just couple 2 models, we plan to couple a number of atmospheric science models including air pollution codes. We also plan to retrieve data directly from scientific instruments like temperature and pressure sensors. Such an application would form an interesting grid application because it would involve integration of a number of separate computational models, running on different computer systems (possibly owned by different organizations) and taking data from real time scientific instruments.

Acknowledgements: The Australian Research Council and Hewlett Packard support this work under an ARC linkage grant. Kommineni was supported by an Australian Postgraduate Research Award.

References

[1] Chan, P. and Abramson, D. “NetFiles: A Novel Approach to Parallel Programming of Master/Worker Applications”, HPC Asia 2001, 24-28 September 2001 • Royal Pines Resort Gold Coast, Queensland, Australia.

[2] Deelman, E., Blackburn, K. et al., "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists," presented at 11th Intl Symposium on High Performance Distributed Computing, 2002.

[3] Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Lazzarini, A., Arbree, A., Cavanaugh, R. and Koranda, S. “Mapping Abstract Complex Workflows onto Grid Environments”, Journal of Grid Computing, Vol. 1, No. 1, pp 9--23, 2003.

[4] Foster, I. and Kesselman, C., Globus: A Metacomputing Infrastructure Toolkit, International Journal of Supercomputer Applications, 11(2): 115-128, 1997.

[5] Foster, I., and Kesselman, C. (editors), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, USA, 1999.

[6] GriPhyN 2003, www.griphyn.org

[7] McGregor, J. L., Nguyen, K. C., and Katzfey, J. J. Regional climate simulations using a stretched-grid global model. In: Research activities in atmospheric and oceanic modelling. H. Ritchie (ed.). (CAS/JSC Working Group on Numerical Experimentation Report; 32; WMO/TD - no. 1105) [Geneva]: WMO. p. 3.15-3.16, 2002.

[8] McGregor, J.L., Walsh, K.J. and Katzfey, J.J. Nested modelling for regional climate studies. In: A.J. Jakeman, M.B. Beck and M.J. McAleer (eds.), Modelling Change in Environmental Systems, J. Wiley and Sons, 367–386, 1993.

[9] Robert van Engelen, “The gSOAP toolkit 2.0.”, Technical report, Florida State University, http://www.cs.fsu.edu/~engelen/soap.html, 2001.

[10] Sun Microsystems, “Web Services Made Easier: The Java APIs & Architectures for XML”, 2002, http://java.sun.com/webservices/white/ & http://java.sun.com/webservices/webservicespack.html

[11] Walsh, K.J. and McGregor, J.L. January and July climate simulations over the Australian region using a limited-area model. J. Climate, 8 (10), 2387–2403, 1995.

Click Here for Full Article

Influence of savanna fire on Australian monsoon season precipitation and circulation as simulated using a distributed computing environment

Amanda H. Lynch,[1] David Abramson,[2] Klaus Görgen,[3] Jason Beringer,[1] and Petteri Uotila[1]

Received 3 June 2007; revised 23 July 2007; accepted 17 August 2007; published 16 October 2007.

[1] Fires in the Australian savanna have been hypothesized to affect monsoon evolution, but the hypothesis is controversial and the effects have not been quantified. A distributed computing approach allows the development of a challenging experimental design that permits simultaneous variation of all fire attributes. The climate model simulations are distributed around multiple independent computer clusters in six countries, an approach that has potential for a range of other large simulation applications in the earth sciences. The experiment clarifies that savanna burning can shape the monsoon through two mechanisms. Boundary-layer circulation and large-scale convergence is intensified monotonically through increasing fire intensity and area burned. However, thresholds of fire timing and area are evident in the consequent influence on monsoon rainfall. In the optimal band of late, high intensity fires with a somewhat limited extent, it is possible for the wet season to be significantly enhanced. **Citation:** Lynch, A. H., D. Abramson, K. Görgen, J. Beringer, and P. Uotila (2007), Influence of savanna fire on Australian monsoon season precipitation and circulation as simulated using a distributed computing environment, *Geophys. Res. Lett.*, *34*, L20801, doi:10.1029/2007GL030879.

1. Introduction

[2] Dry season fires are one of the largest natural and anthropogenic disturbances in the tropical savannas. Between 40 to 75% of global savannas were burned annually between 1975 and 1980 [*Hao et al.*, 1990]. Savanna fires consume three times as much dry matter annually as the burning of tropical forests (e.g., 3690 Tg dm yr^{-1} for savanna versus 1260 Tg dm yr^{-1} for tropical forests [*Andreae*, 1991]) although recovery of the biomass is an order of magnitude more rapid. The Australian tropical savanna is one of the world's largest, with vast tracts affected by fire each year, particularly in deliberate burning by pastoralists, Aboriginal land holders and conservation managers – an estimated 244,000 km^2 of the total area of northern Australia was affected in 1997 [*Russell-Smith et al.*, 2000].

[3] Fires across this landscape have impacts on the regional water, energy and carbon dioxide exchanges [*Beringer et al.*, 2003] through altered vegetation composition and reduced surface albedo, increased available energy for partitioning into the convective fluxes, and increased substrate heat flux into the soil. In addition, the aerodynamic and biological properties of the ecosystem can change, affecting surface-atmosphere interaction [*Beringer et al.*, 2007]. Local tethered balloon profile measurements [*Wendt et al.*, 2007] and idealized computer sensitivity studies [*Görgen et al.*, 2006; *Pitman and Hesse*, 2006] have demonstrated intensified boundary layer processes in fire-affected areas and have led some to suggest impacts extending to the regional scale [*Miller et al.*, 2005]. However, these impacts, and their potential for the modification of the Australian monsoon regime, have not been systematically quantified. Here we address that quantification using a comprehensive experimental design made possible through the use of software that allows the implementation of legacy code on a global computational grid.

2. Methods

[4] Quantifying the impacts of fire in savanna regions is challenging because fire and its effects span many temporal and spatial scales. Moreover, there is high interannual variability. Hence, this question requires the application of a high resolution climate model which allows the representation of the effects of savanna fire across a range of manifestations through multiple realizations of the model.

[5] We represented savanna fires in the Conformal-Cubic Atmospheric Model (C-CAM) from the Australian Commonwealth Scientific and Industrial Research Organization (CSIRO), which is a global hydrostatic atmospheric model coupled to a land surface model with a variable-resolution conformal-cubic grid [*McGregor and Dix*, 2001]. This grid provides a resolution of approximately 65 km over Australia extending to about 800 km on the far side of the globe. Far-field atmospheric nudging is achieved using the National Center for Environmental Prediction/National Center for Atmospheric Research (NCEP/NCAR) reanalysis data [*Kalnay et al.*, 1996] with an e-folding time of 96 hours, resulting in relatively weak constraints on the high-resolution part of the model domain [*Görgen et al.*, 2006].

[6] A fire/re-growth scheme has been implemented in the existing C-CAM land surface model. The surface properties modified by fire and the subsequent vegetation re-growth

[1]Climate Program, Monash Sustainability Institute and School of Geography and Environmental Science, Monash University, Clayton, Victoria, Australia.
[2]Clayton School of Information Technology, Monash University, Clayton, Victoria, Australia.
[3]Département Environnement et Agro-Biotechnologies, Centre de Recherche Public – Gabriel Lippmann, Belvaux, Luxembourg.

Copyright 2007 by the American Geophysical Union.
0094-8276/07/2007GL030879$05.00

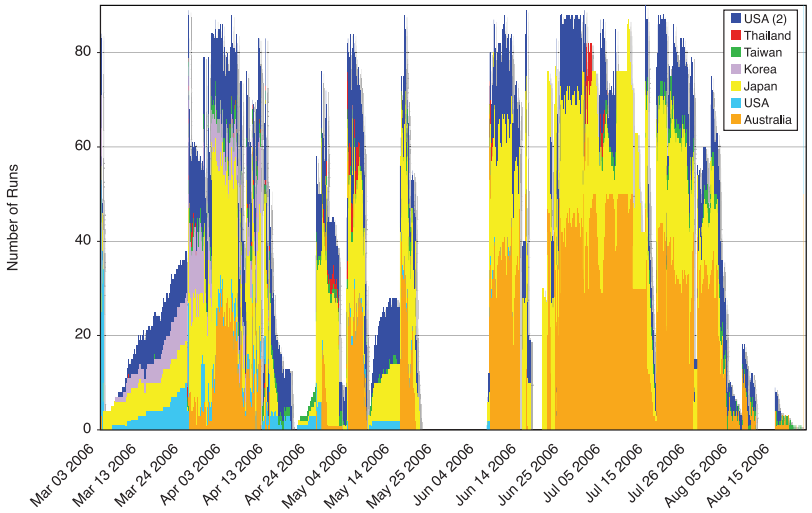


Figure 1. Number of jobs executed by computational resources in each country participating in the experiment over the PRAGMA grid (www.pragma-grid.org).

are defined in the model by four parameters which are based on the observed fire regime [*Russell-Smith et al.*, 2003]. First, the intensity of the fire in any one grid cell ranges from 10% to 100% of the maximum possible biomass that can be destroyed, on the basis of fire intensities associated with an energy release between 10^3 and 10^4 kWm^{-1} [*Chafer et al.*, 2004]. Secondly, the spatial distribution of fire-affected grid cells is defined within a range from 10% to 100% of the maximum possible area, based on the observed area burnt between 1997 and 2001. Thirdly, the temporal range for the timing of the fire event lies between 1 May and 30 November, which is followed a few weeks after by the average monsoon onset date. Finally, the length of the regrowth period is allowed to vary between 35 and 140 days.

[7] In order to quantify the relative importance of each of the four forcing parameters (area, intensity, timing and length of regrowth), a multifactorial approach that allows the analysis of both the impacts of individual forcings and the interactions between forcings is required. *Rivers and Lynch* [2004] developed just such a method using a factorial experimental strategy based on the work of *Henderson-Sellers* [1993] for assessing the effects of the changing land surface on early Holocene climate in Beringia using a series of multi-year ensemble perturbation experiments that varied all forcing parameters simultaneously. This approach has been adopted here by varying all four forcing parameters simultaneously using a Latin hypercube sampling to define 90 individual experiments, each of 21 years duration (1979–1999) and each having a 5 year fire-free spin up period.

[8] Using a single sequential high end workstation would require more than 10 years of wall clock time to complete this experiment. Clearly this is unacceptable. However, the individual realizations can be executed in parallel and it is possible to reduce the execution time substantially using this technique. Rather than using a single 90 processor cluster (which was not available), we distributed the computations across multiple independent clusters located in Australia, Japan, Korea, Taiwan, Thailand and US. These have been provided as part of a multi-country Grid computing initiative called PRAGMA (the Pacific Rim Grid and Middleware Assembly).

[9] Running such a large scale experiment over distributed resources is very time consuming and error prone without adequate software support. In response, *Abramson et al.* [2000] have developed a software tool called Nimrod/G, which automates the execution of parameter sweep and search applications, such as the design developed here, over global computational Grids. Nimrod/G manages the complexity of running the experiment, handles network and resource failures, and gives the appearance of a single supercomputer.

[10] Our experiment ran for an elapsed time of 170 days – significantly less than 10 years – with an average of 37 processors in use across that period (Figure 1). Of the 90 original experiments, 85 executed successfully, and 5 were terminated due to instabilities in the model.

3. Response of the Monsoon

[11] The area-total monsoon precipitation over northern Australia responds, with a statistically significant increase, to increases in the fire intensity, the lateness of the timing and the area burned in the preceding dry season. The maximum increase of the simulations performed, due to the combination of all four parameters, is 1.6 mm day^{-1} over an average precipitation rate of 5.1 mm day^{-1} (a 31% increase in average monsoon season precipitation) (Figure 2a). This maximal response occurs in the simulation that specifies a fire intensity of 72%, a fire area of 90%, and a date of fire of the 5th November, around 6 weeks before typical monsoon onset [*Hendon and Liebmann*, 1990]. The length of the regrowth period has a limited impact on monsoon precipitation, accounting for only around 9% of the total variance in monsoon precipitation. In contrast, the fire intensity accounts for around 15% of the variance, the fire area accounts for almost 18% of the variance, and the fire timing accounts for almost 58% of the variance in monsoon precipitation.

[12] Results from the 85 multiple realizations allowed the development of a statistical model [*Lynch et al.*, 2001] of the response metrics (such as monsoon onset and intensity) to variations in the four driving parameters. The statistical model allows inter- and extra-polation from the simulations that were successfully performed to postulate the maximal response. This model suggests that the largest increase in monsoon rainfall in response to previous dry season fires is dependent upon a maximum possible fire intensity occurring as late as possible during the dry season, but does not support the largest possible fire area. This is due to the need for sufficient vegetated area to remain to boost the moisture available for convective activity. This limit occurs at around 90% of the total possible burned area.

[13] The interaction between monsoon season precipitation, fire intensity and the timing of the fire is of particular interest to decision-makers. High intensity fires that burn both the over-storey and the undergrowth in the early dry season (up to around mid-July) have limited impact on monsoon precipitation because atmospheric moisture is limited (Figure 2a). Conversely, late dry season fires have a small influence on precipitation unless the fire is also very intense. Fire area and timing also interact such that a large fire area has effectively no impact during early dry season fires, but fires of any area between around 40% and 90% had a strong influence on monsoon precipitation if they occurred during the late dry season. The relationship between monsoon precipitation and fire area shows less variability than that between monsoon precipitation and fire intensity. The variability in monsoon rainfall is due specifically to variations in uplift intensity (see Figure 2b, discussed below) rather than moisture availability.

[14] The impact of dry season fire intensity on monsoon season precipitation is due to the interaction between the generation of uplift due to surface heating and the availability of moisture. Fires result in decreased albedo, increased net radiation, increased Bowen ratio (surface heating) and enhanced vertical uplift. The area over which enhanced vertical motion exceeds a threshold value of 1.5 Pa s^{-1} (accumulated over three model levels in the upper boundary layer) increases with both intensity (Figure 2b) and area of fire. The influence of the timing of the fire on vertical motion is rather small, as the intensification of boundary layer processes is not driven by moist processes. However, the timing of the fire becomes important in the simulation of moist parameters, including latent heat fluxes, precipitable water in the tropospheric column, and convective cloud (Figure 2c). There is insufficient available moisture for fires to influence these variables in the early dry season, but during monsoon buildup a combination of

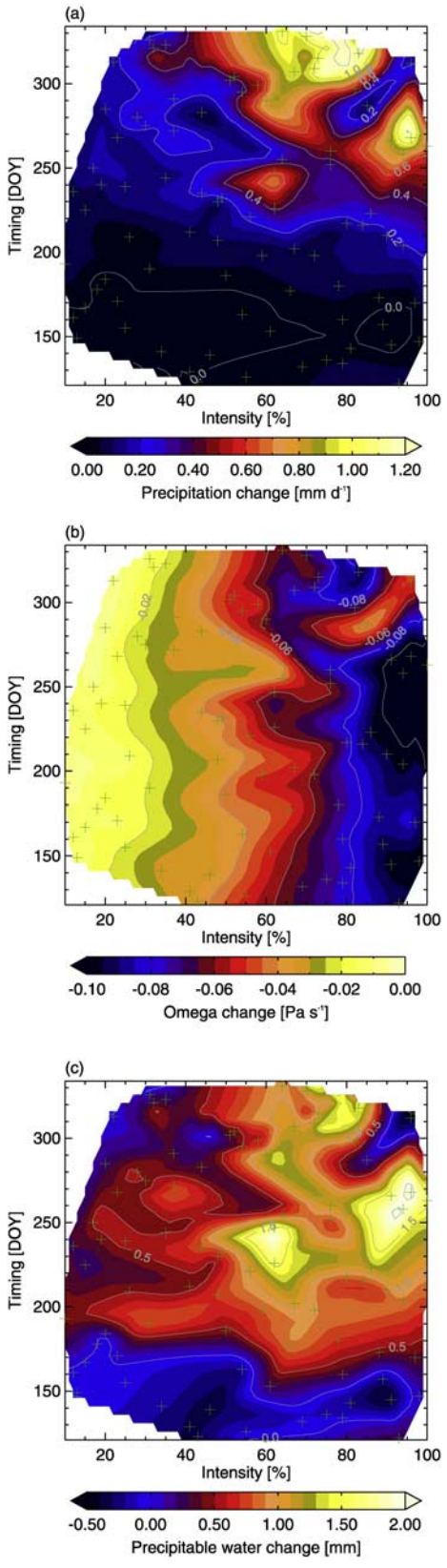


Figure 2. Simulated differences (scenario minus reference simulation) in response to variation in fire intensity and the date of fire onset. Shown are differences in (a) average precipitation [mm d^{-1}], (b) vertically averaged boundary layer uplift [Pa s^{-1}], spatially averaged over those grid elements that satisfy a threshold criterion (below 1.5 Pa s^{-1}), and (c) vertically integrated precipitable water [mm]. The differences of the spatial averages over Australia's land area north of Tropic of Capricorn are averaged over the first 60% of the regrowth period over the 20 years of each experiment. The crosses indicate the population of the simulation space. The contoured response metric is then Gaussian low pass filtered linear interpolations based on Delauney triangulations.

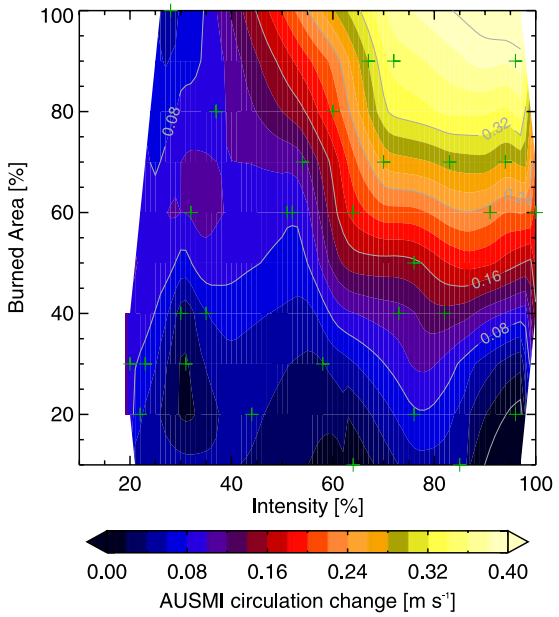


Figure 3. Simulated differences (scenario minus reference simulation) in response to variation in fire intensity and burned area. Shown are differences in the mean zonal wind speed [m s^{-1}] at 850 hPa. Spatial averages of zonal winds are calculated from 0° to 10°S and 120°E to 150°E, the zone of monsoonal reversal of the flow in and out of Australia. Only experiments with a timing of the fire-event later than the Julian day 250 (7 September) are considered, as indicated by crosses.

enhanced convective activity with increased atmospheric moisture leads to both an advancement of monsoon onset and increased precipitation intensity, and hence an increase in total monsoon season precipitation.

[15] Given the large scale influence of fire on surface energy balance, albedo, vertical motion and moist processes, we hypothesized that the simulated changes may have an impact on regional scale monsoon circulation patterns such as that postulated by *Johnson et al.* [1999]. To test this we used an indicator of large-scale circulation changes called the Australian Monsoon Index (AUSMI) [*Wang et al.*, 2004], which is obtained by averaging the daily mean 850 hPa zonal wind speed from the equator to 10°S and from 120°E to 150°E. Earlier studies [*Görgen et al.*, 2006] using single fire realizations have found that fires have a limited impact, if any, on the large scale development of the monsoon regime, with responses that are temporally and spatially restricted to fire-affected areas. However, in this more complete experimental design, we have found a significant influence of fire intensity and area on the AUSMI when consideration is restricted to late dry season fires (Figure 3). For example, the maximum value of AUSMI is 0.43 ms^{-1} for a high intensity, late dry season fire, compared to the average of all scenarios of 0.03 ms^{-1}. As for the local impacts, the moisture availability later in the year is a requirement for a statistically significant perturbation to the monsoon circulation. In contrast, fires that occur before September have no discernable influence on the large scale monsoon circulation. Thus, across the experiments performed, fire timing accounts for almost 85% of the variance in AUSMI. Nevertheless, changes in both fire intensity and area burned account for 6% of the variance respectively, and statistical modeling indicates that the area threshold noted for maximum monsoon rainfall impact is not evident in the influence on circulation.

[16] These responses are certainly model dependent, due to variations in sensitivity amongst climate models, the impacts of grid resolution and parameterization choice, and the possible constraining effect of far field nudging. However, the technical innovation that supports this experimental design is highly scalable, and hence an extension of this work will be to consider multi-model ensembles. In this context, these experiments suggest that savanna burning can shape monsoon response through two mechanisms. The first, and primary, mechanism is the influence on the intensity of surface heating, which has a direct effect on the boundary layer circulation and thence large scale convergence. The secondary mechanism is the influence of these changes on convection and monsoon precipitation – evident is a threshold of fire timing on the one hand and fire area on the other. Under the appropriate condition of sufficient available moisture combined with enhanced convergence and uplift, the subsequent wet season is indeed significantly enhanced.

[17] **Acknowledgments.** This research was supported by the Australian Research Council though grants FF0348550 and DP034474. The support of the Cooperative Research Centre for Enterprise Distributed Systems, the Department of Communications, Information Technology and the Arts under a GranetNet grant, and the Australian Partnership for Advanced Computing are also gratefully acknowledged. The support of the reviewers to improve the manuscript is recognized and appreciated.

References

Abramson, D., J. Giddy, and L. Kotler (2000), High performance parametric modeling with Nimrod/G: Killer application for the global grid? Paper presented at International Parallel and Distributed Processing Symposium, Inst. of Electr. and Electr. Eng., Cancun, Mexico.

Andreae, M. O. (1991), Biomass burning: Its history, use and distribution and its impact on environmental quality and global climate, in *Global Biomass Burning: Atmospheric, Climatic and Biospheric Implications*, edited by J. S. Levine, pp. 3–21, MIT Press, Cambridge, Mass.

Beringer, J., et al. (2003), Fire impacts on surface heat, moisture and carbon fluxes from a tropical savanna in north Australia, *Int. J. Wildland Fire*, *12*, 333–340.

Beringer, J., L. B. Hutley, N. J. Tapper, and L. J. Cernusak (2007), Savanna fires and their impact on net ecosystem productivity in North Australia, *Global Change Biol.*, *13*, 990–1004, doi:10.1111/j.1365-2486.2006.01334.x.

Chafer, C. J., M. Noonan, and E. Macnaught (2004), The post-fire measurement of fire severity intensity in the Christmas 2001 Sydney wildfires, *Int. J. Wildland Fire*, *13*, 227–240.

Görgen, K., A. H. Lynch, A. G. Marshall, and J. Beringer (2006), The impact of abrupt land cover changes by savanna fire on northern Australian climate, *J. Geophys. Res.*, *111*, D19106, doi:10.1029/2005JD006860.

Hao, W. M., M.-H. Liu, and P. J. Crutzen (1990), Estimates of annual and regional releases of CO_2, and other trace gases to the atmosphere from fires in the tropics, based on FAO statistics for the period 1975–1980, in *Fire in the Tropical Biota*, *Ecol. Stud.*, vol. 84, edited by J. G. Goldammer, pp. 440–462, Springer, New York.

Henderson-Sellers, A. (1993), A factorial assessment of the sensitivity of the BATS land-surface parameterization scheme, *J. Clim.*, *6*, 227–247.

Hendon, H. H., and B. Liebmann (1990), A composite study of onset of the Australian summer monsoon, *J. Atmos. Sci.*, *47*, 2227–2240.

Johnson, B. J., et al. (1999), 65,000 years of vegetation change in central Australia and the Australian summer monsoon, *Science*, *284*, 1150–1152.

Kalnay, E., et al. (1996), The NCEP/NCAR 40-year reanalysis project, *Bull. Am. Meteorol. Soc.*, *77*, 437–472, doi:10.1175/1520-0477.

Lynch, A. H., S. McIlwaine, J. Beringer, and G. B. Bonan (2001), An investigation of the sensitivity of a land surface model to climate change using a reduced form model, *Clim. Dyn.*, *17*, 643–652.

McGregor, J. L., and M. R. Dix (2001), The CSIRO conformal-cubic atmospheric GCM, in *IUTAM Symposium on Advances in Mathematical Modelling of Atmosphere and Ocean Dynamics*, edited by P. F. Hodnett, pp. 197–202, Kluwer, Dordrecht, Netherlands.

Miller, G., et al. (2005), Sensitivity of the Australian Monsoon to insolation and vegetation: Implications for human impact on continental moisture balance, *Geology*, *33*, 65–68.

Pitman, A. J., and P. P. Hesse (2006), The significance of large-scale land cover change on the Australian palaeomonsoon, *Quat. Sci. Rev.*, *26*, 189–200, doi:10.1016/j.quascirev.2006.06.014.

Rivers, A. R., and A. H. Lynch (2004), The influence of land surface transitions on the local climate of Beringia in the early Holocene, *J. Geophys. Res.*, *109*, D21114, doi:10.1029/2003JD004213.

Russell-Smith, J., G. Allan, R. Thackway, R. Rosling, and R. Smith (2000), Fire management and savanna landscapes in northern Australia, in *Fire and Sustainable Agricultural and Forestry Development in Eastern Indonesia and Northern Australia*, *ACIAR Proc.*, vol. 91, edited by J. Russell-Smith et al., pp. 95–101, Aust. Cent. for Int. Agric. Res., Canberra, A. C. T.

Russell-Smith, J., et al. (2003), Contemporary fire regimes of northern Australia, 1997–2001: Change since Aboriginal occupancy, challenges for sustainable management, *Int. J. Wildland Fire*, *12*, 283–297.

Wang, B., I.-S. Kang, and J.-Y. Lee (2004), Ensemble simulations of Asian-Australian-Monsoon variability by 11 AGCMs, *J. Clim.*, *17*, 803–818.

Wendt, C., J. Beringer, N. Tapper, and L. Hutley (2007), Local boundary-layer development over burnt and unburnt tropical savanna: An observational study, *Boundary Layer Meteorol.*, *124*, 291–304.

D. Abramson, Clayton School of Information Technology, Monash University, Clayton, VIC 3800, Australia.

J. Beringer, A. H. Lynch, and P. Uotila, Climate Program, Monash Sustainability Institute and School of Geography and Environmental Science, Monash University, Clayton, VIC 3800, Australia. (amanda.lynch@arts.monash.edu.au)

K. Görgen, Département Environnement et Agro-Biotechnologies, Centre de Recherche Public–Gabriel Lippmann, L-4422 Belvaux, Luxembourg.

An Application of Optimisation for Passive RF Component Design

Andrew Lewis, Seppo A. Saario, David Abramson* and Jun W. Lu
Griffith University, Brisbane QLD 4111, Australia
*Monash University, Melbourne VIC 3168, Australia

***Abstract*—The solution of computational electromagnetic simulations is integral to the design process. As higher performance computers become more available, the application of optimisation techniques to reduce design times becomes more feasible. This paper presents the application of Parallel BFGS and Adaptive Simulated Annealing in minimising the transmission through a ceramic bead suppressor on a straight wire transmission line.**

I. Introduction

A high permittivity ceramic bead on a straight wire transmision line can be used to suppress propagating cable currents at RF frequencies [1]. In our application, we wish to have minimum transmission S_{21} through the bead at 1.0GHz. The FDTD method is used to perform a full-wave analysis of the cable structure. We present the application of an Adaptive Simulated Annealing (ASA) algorithm implemented by Lester Ingber [2] and a parallel implementation of the quasi-Newton BFGS algorithm (P-BFGS) [3] developed at the Queensland Parallel Supercomputing Facility (QPSF) in the design of such beads.

In the optimisation process, the convergence rate and susceptibility to local minima determine the effectiveness of a technique. It is well known that ASA is very good at finding a global minimum in complex problems. However, ASA is generally computationally intensive and requires careful selection of optimisation parameters to guarantee finding a global minimum.

For continuous functions, gradient descent methods can be particularly effective. The algorithm selected is based on the quasi-Newton BFGS method. Because of the computationally intensive nature of the FDTD simulation a parallel implementation of the BFGS algorithm was utilised.

II. Results

Both ASA and P-BFGS algorithms were used to optimise the ceramic bead dimensions (thickness, length) and permittivity. A cost function to drive the optimisation algorithms was defined as the average S_{21} over the 990-1010MHz band, subtracted from 50. The cost function is defined as: $C(p_k) = Avg(S_{21}) - 50$. An isosurface of the cost function is shown in Figure 1.

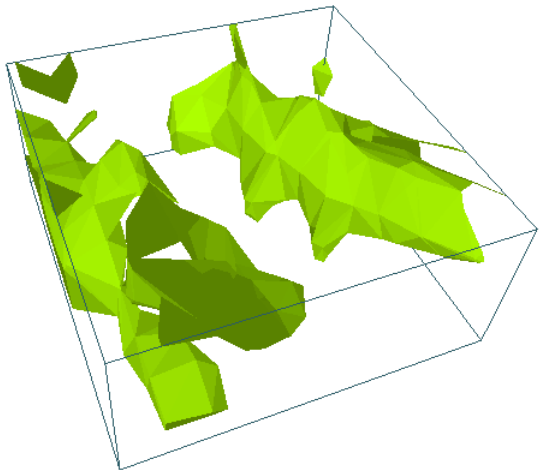


Fig. 1. Cost function isosurface at 25 (S_{21} = -25dB)

Optimisation results from both algortithms were not identical. After 143 function evaluations, the ASA solution defined a bead of $\varepsilon_r = 52$, $O.D. = 20mm$ and $length = 57.5mm$ to have an S_{11} of -10.3dB and S_{21} of -49dB. P-BFGS yielded a bead of $\varepsilon_r = 41.8$, $O.D. = 40mm$ and $length = 102mm$ to have an S_{11} of -14.7 dB and S_{21} of -56.6 dB after 44 steps on an 8 processor machine.

III. Conclusion

The effectiveness of two types of optimisation algorithms was investigated for the optimisation of the high permittivity ceramic suppression beads. Both ASA and P-BFGS provided workable solutions. Further investigations are proceeding into problems with each type of optimisation algorithm. Based on the observed behaviour with a relatively complex problem, possible hybrid schemes are being considered.

References

[1] S. A. Saario, D. V. Thiel and Jun W. Lu, "Application and optimisation of high permittivity ceramic beads for RF isolation on straight wire transmission lines" *CSIRO, Symposium on Antennas* , 17-18 Feb. 1999, Sydney, Australia.

[2] Lester Ingber, "Lester Ingber's Code and Reprint Archive" *http://www.ingber.com.*

[3] A. Lewis, D. Abramson and R. Simpson, "Parallel Non-Linear Optimization: Towards the Design of a Decision S upport System for Air Qaulity Management", *Proc. ACM/IEEE SC97 Conference*, San Jose, November 15-21, 1997.

Incorporating local Ca^{2+} dynamics into single cell ventricular models

Anna Sher[2], David Abramson[1], Colin Enticott[1], Slavisa Garic[1], David Gavaghan[2], Denis Noble[3], Penelope Noble[3] and Tom Peachey[1],

[1] Faculty of Information Technology,
Monash University,
Clayton, 3800, Victoria, Australia

[2] Comp. Biology Group,
Oxford University Computing Laboratory,
Oxford OX1 3QD, UK

[3] Department of Physiology,
Anatomy and Genetic,
Oxford University
Oxford OX1 3PT, UK

Abstract. Understanding physiological mechanisms underlying the activity of the heart is of great medical importance. Mathematical modeling and numerical simulation have become a widely accepted method of unraveling the underlying mechanism of the heart. Calcium (Ca^{2+}) dynamics regulate the excitation-contraction coupling in heart muscle cells and hence are among the key players in maintaining normal activity of the heart. Many existing ventricular single cell models lack the biophysically detailed description of the Ca^{2+} dynamics. In this paper we examine how we can improve existing ventricular cell models by replacing their description of Ca^{2+} dynamics with the local Ca^{2+} control models. When replacing the existing Ca^{2+} dynamics in a given cell model with a different Ca^{2+} description, the parameters of the Ca^{2+} subsystem need to be re-fitted. Moreover, the search through the plausible parameter space is computationally very intensive. Thus, the Grid enabled Nimrod/O software tools are used for optimizing the cell parameters. Nimrod/O provides a convenient, user-friendly framework for this as exemplified by the incorporation of local Ca^{2+} dynamics into the ventricular single cell Noble 1998 model.

Key words: Cardiac Cells, Mathematical modeling, Parameter optimization, Grid Computing.

1 Introduction

Researchers have been developing complex models of cardiac cells for many years, in an attempt to explore the detailed physiology and operation of the heart. Ultimately, the goal is to produce better treatment strategies and to develop novel drugs for treating heart disease. This case study concerns the detailed modeling of particular ion channels in heart muscle cells.

The key physiological function of the heart is to pump blood around the living organism. This function is enabled by the spread of electrical excitation through the cardiac tissue and contraction of the cardiac muscles. On the single-cell level (a myocyte), the mechanisms of excitation-contraction coupling are closely regulated by calcium ion (Ca^{2+}) dynamics. Ca^{2+} entering the cell triggers the release of Ca^{2+} from the sarcoplasmic reticulum (SR) which is the organelle that stores calcium. The resulting rise of intracellular Ca^{2+} (Ca_i) activates the contraction of the cell. This phenomenon is known as Ca^{2+}-induced Ca^{2+} release (CICR). Local Ca^{2+} dynamics are characterized by the interactions within localized microdomains (known as dyadic spaces) between L-type Ca^{2+} channels (LCCs) located on the Transverse–tubules (T–tubules), which are deep invaginations of the membrane into the cell, and closely opposed Ca^{2+} release channels (known as ryanodine receptors, RyRs) located on the sarcoplasmic reticulum (Fig. 1). The sarcoplasmic reticulum is an extensive and well organized network, that repeatedly comes in contact with each T-tubule, so that the number of dyadic spaces throughout the cell has been estimated to be of the order 50,000 – 300,000.

The local Ca^{2+} release mechanisms are essential to reproduce the characteristic properties of the excitation-contraction coupling such as high gain and graded Ca^{2+} release. However, most existing single cell models lack the description of the biophysical nature of local Ca^{2+} dynamics. In this paper we present a methodology of how local Ca^{2+} dynamics can be efficiently incorporated into a single cell model of ventricular myocyte in order to produce a biophysically accurate cell model. The two stages involved are (i) development of the Ca^{2+} subsystem and (ii) its incorporation into a single cell model. The first stage is the generation of the local control CICR models (also known as the coupled LCC-RyR models) such as, for instance, the ones that have been developed by Hinch et al. [2] and Greenstein et al. [1]. The second stage, which is the focus of this paper, involves the incorporation of the coupled LCC-RyR models into a single cell model. Specifically, the steps are as follows:

- The equations that describe Ca^{2+} dynamics in the original single cell model (e.g. Noble 1998 model[1] [4] (Fig. 2)) are substituted by equations of the biophysically detailed Ca^{2+} subsystem (e.g. baseline 40-state coupled LCC-RyR Greenstein 2006 model [1] (Fig. 1)), provided that units are modified accordingly;
- The parameters of the newly obtained single cell model are refitted. This is done to ensure that the newly obtained single cell model, which contains the replaced Ca^{2+}

[1] The Noble 1998 model is extensively used by various researchers and, thus, it is important to assess the effect of replacing the existing phenomenological description of Ca^{2+} in the dyadic space and Ca^{2+} related currents with the local, biophysically sound Ca^{2+} dynamics. Therefore, the Noble 1998 model is chosen as the case study.

subsystem, is capable of reproducing the data of the original model. In particular, the specific aim is to fit the dynamics of the Ca^{2+} of the newly developed whole-cell model either to the dynamics of the Ca^{2+} of the original model (e.g. Noble 1998) and/or to the available experimental data (e.g. Ca_i transient, IV curves, tail currents recorded from the voltage-clamp experiments, etc.). To achieve this, we need to optimize the parameters of the Ca^{2+} subsystem, or, in other words, to solve an inverse problem.

In this paper we demonstrate how the novel Grid computing tools allows the incorporation of local Ca^{2+} dynamics into the existing cellular models at a low computational cost. The optimization methods that are used require repeated evaluation of the models, and thus the time required to compute the optimal model parameters can be very long. The computational Grid can be exploited to speed the execution by delivering a large number of processors. The Grid enabled Nimrod/O tool that we use in this experiment incorporates a range of non-linear optimization methods, and these can be used to optimize the cell parameters accordingly. Section 2 briefly introduces the Grid and the Nimrod tools. Section 3 discusses challenges and results of incorporating local calcium dynamics on the example of the ventricular single cell Noble 1998 model.

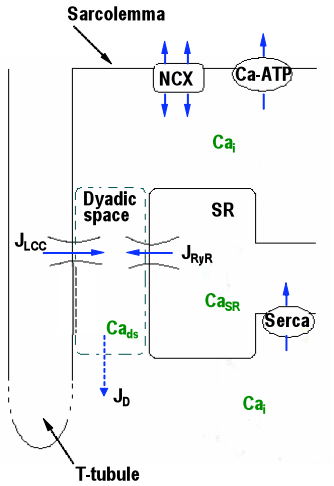


Fig 1. Local Ca^{2+} dynamics of Greenstein et al. 2006 [1] model. Arrows represent the direction in which Ca^{2+} flows. Ca_i, Ca_{ds} and Ca_{SR} denote intracellular, dyadic and SR Ca^{2+} respectively. The diagram illustrates the local control theory: LCCs and RyRs contribute to local J_{LCC} and J_{RyR} fluxes respectively within the dyadic space, J_D represents the diffusion of Ca^{2+} out of the dyad into the bulk myoplasm, SERCA re-uptakes Ca^{2+} back into the SR, Ca^{2+}-ATPases and NCXs pump Ca^{2+} out of the cell.

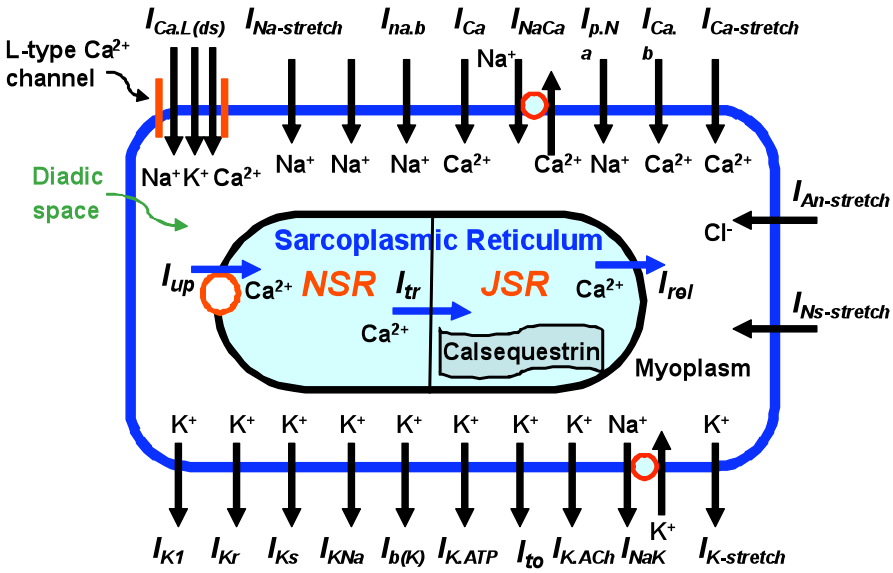


Fig 2. Schematic representation of the single cell Noble et al. 1998 [3] model (adapted from a diagram at `www.cellml.org` model repository). Arrows represent the direction in which the ions flow and the label of the corresponding ionic current is located above or bellow the arrow. The model includes four Ca^{2+} compartments which are the intracellular, dyadic, network SR (NSR) and junctional SR (JSR). This model is an example of the deterministic non-common pool ventricular model that uses phenomenological description of Ca^{2+} in the dyadic space and Ca^{2+} related currents. While such a model succeeds in producing graded SR Ca^{2+} release, it lacks the mechanistic description of local SR Ca^{2+} release, i.e. the stochastic interaction between LCCs and RyRs within the dyadic spaces.

2 Grid Computing

The Grid provides a general platform for integrating computation, data and instruments [8]. It serves as the infrastructure for implementing novel applications, particular in science and engineering.

In particular, “computational” Grids have emerged as a viable platform for delivering on-demand access to a range of very high performance machines. Whilst it may not be possible to gain access to sufficient resources at any single site, computational Grids can aggregate a number of otherwise separate resources into a single large super-computer. Such a virtual machine, or testbed, is an ideal base for simulating complex systems using computational models because the resources can be assembled at a period of peak demand and then released for use when not required. Such platforms have the potential to offer very cost effective solutions, leveraging everything from spare cycles on high end machines through to large pools of inexpensive desktops that are idle.

In spite of the enormous progress in building operation Grids, and the significant effort in developing middleware, assembling such a testbed on demand is difficult. Most Grids are built from different components, and this resource heterogeneity is a fact of life. Likewise, Grids are built across multiple administrative and security domains, posing problems for aggregating them into a single virtual machine. Lack of a single owning organization also means that resource scheduling becomes complex – no single job scheduler can guarantee access to sufficient computational power, making it difficult to deliver the guaranteed levels of service. Importantly, Grid application users don't want to know about the complexity of the underlying fabric, and wish to concentrate on their domain science.

Difficulty in using the Grid is not a hypothetical concern. Currently, very few scientists use the Grid routinely, and instead rely on local resources, which are under their control. This means that the scale and nature of the work is limited. Until we can make it easier to use, the Grid will never be adopted by more than the most hardy or desperate users!

Over the years we have developed a strategy for delivering the high levels of performance, and have built software tools that make it easy for scientists to leverage the computational power of the Grid. Specifically, the Nimrod family of tools allows a non-expert to specify large computational experiments using legacy software, and execute these over a range of Grid resources. Nimrod is not a single tool: it incorporates a component that distributes computations to the resources (Nimrod/G) [5][7]; a component that searches for "good" solutions using non-linear optimization algorithms (Nimrod/O) [4][6]; and a component that helps evaluate which parameter settings are important using experimental design (Nimrod/E). Most aspects of Nimrod have been written about extensively over the years, so we will only provide a cursory overview in Section 2.1 of the paper.

2.1 The Nimrod Tool family

Figure 3 shows the architecture of the Nimrod tool family and the interaction between the major components. Typically, users interact through a Web browser using the Nimrod portal. This single point of presence then directs traffic to one of three different components – Nimrod/G which support parameter studies and distributes the computations to the Grid, Nimrod/O which performs optimization and Nimrod/E which uses experimental design techniques to scope parameter studies. Importantly, each of these components acts either as a user level tool, or as middleware, depending on the client use. For example, Nimrod/G can interact directly with users using a Web enabled interface, or can provide services to other software (such as Nimrod/E, Nimrod/O) via an API. Each of the applications discussed here leverages different aspects of the tools. In many cases, they used Nimrod/G to perform a crude sweep of the overall parameter space, and then launched Nimrod/O to refine the solutions. Nimrod/E is a fairly new development, and whilst it has been used in the cardiac modeling work, we do not have results at this stage.

An important aspect of the tool family is that they share a common specification language – which is written in a text document called a "plan" file. This file contains details of the parameters and how to invoke the application, and is typically quite

small. Over the years, we have expanded the plan file to allow more complex workflows to be specified [9], however, in the simplest form a single application is run many times. Nimrod/O plan files contain some additional information about which heuristics to use This specifies the optimization algorithm, or algorithms, and associated settings. For example, the file may specify simulated annealing and the associated cooling regime. Starting points for iterative algorithms are also specified as Nimrod/O can perform multiple concurrent searches. The Nimrod/E plan file contains information about which parameter combinations are to be estimated, and which are assumed negligible.

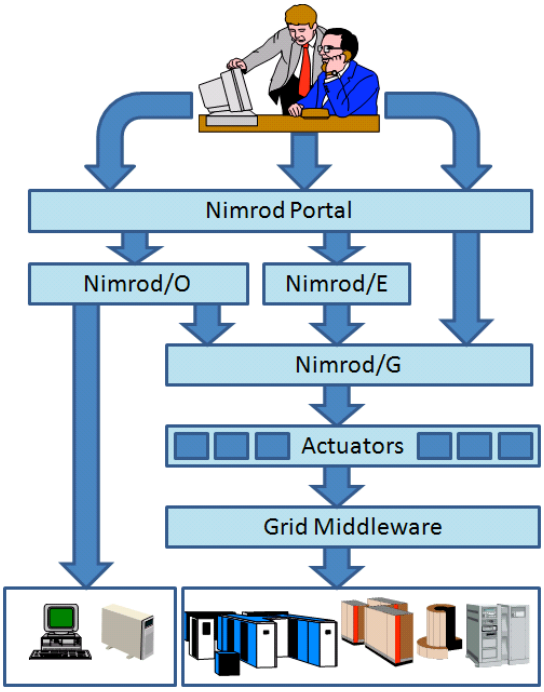


Fig 3. The Nimrod tool chain

2.2 Nimrod methodology

Each of the case studies discussed in the next section adopted the same overall methodology – regardless of which Nimrod tool was used. The following steps summarize this process:

1. Testbed construction. The use must decide which resources will be included in the grid testbed, and configure Nimrod to use these. The Nimrod portal provides a number of high level interfaces for making this fairly easy. Nimrod assumes that

users already have accounts (and the necessary authentication) on each of the testbed resources.
2. Software preparation. Here the applications are compiled and tested on each of the Grid resources. This can either be performed manually by logging into each of the different remote resources, or by using a tool like Distant [9] which manages the process through a single user oriented client. Even when configured manually, it is possible to prepare the application binary on one machine, and use Nimrod to distribute it to similar resources before execution.
3. Determine which Nimrod tool to use. As discussed, Nimrod has a number of different components. The user must select the most appropriate component, depending on whether a complete, partial or guided search is required.
4. Describe how to execute the application, and which files are required for input and output. These steps are described in the Nimrod plan file, using a simple declarative language. Nimrod can be instructed to copy input files to each resource, and return output files. Large output files can be left on remote resources for later analysis. Nimrod also managed parameter substitution via command line options or special control files.
5. Determine the parameters and their ranges. This will vary depending on the application requirements, These are then described in the Nimrod plan file using the 'parameter' keyword. Most parameters are independent, however, it is also possible to specify sequences of parameters that create complex workflows [9]. In this paper we use Nimrod/O to compute optimal parameter settings.
6. Execute the experiment. This is usually performed through the Nimrod portal, but it is also possible to use the Nimrod command line tools. Long running experiments can be left unattended, and monitored using the Nimrod monitoring tools.
7. Analyze the results, possibly returning to step 5 to refine the parameter ranges.

3 Incorporating local Ca^{2+} dynamics

The models of the local Ca^{2+} dynamics is a system of ODEs of approximately 30-70 variables with up to 100 parameters. These ODEs do not exhibit stiffness, thus, time integrators such as the forward Euler integrator or a Runge-Kutta 4th order method are appropriate to simulate these Markov models. The results presented below are simulated in Matlab 6.5 using an inbuilt `ode45' solver - a one-step solver based on an explicit Runge-Kutta of 4th and 5th order, that is appropriate for non-stiff problems and has medium accuracy.

Each simulation on a personal laptop (e.g. Toshiba 512 MB RAM, 2 GHz, 60 GB Hard Drive, Windows XP) takes under five minutes, thus the computational resources required to perform one simulation are minimal. However, to optimize the set of parameters in the newly developed Ca^{2+} subsystem, the software tools which perform optimization algorithms within the framework of the distributive computing are essential. In particular, Nimrod/O provides a computationally effective manner of tuning the parameters and examining their effects within the newly developed models. Interestingly, in the case of running the simulations discussed above, the limiting

factor is the number of Matlab licenses available rather than the number of processors. This is an issue that requires consideration by both the community and independent software vendors if the true power of the Grid is to be realized for this class of software. Nimrod/O offers a variety of optimization methods, such as "subdivision search" and downhill type search methods, etc. The simulation results presented below are obtained using the downhill simplex method of Nelder and Mead. The optimal set of parameters, calculated using the simplex method, is obtained by fitting the action potential (AP), Ca_i transient and I_{CaL} current, with the objective function calculated using the least-square approximations.

The direct incorporation of the canine 40-state Greenstein 2006 coupled LCC-RyR model 1 into Noble 1998 guinea pig model (Fig. 2) results in a distorted electrical behaviour of the cell such as a significant second peak in Ca_i transient and a pronounced plateau phase in an action potential (compare dashed and dotted curves in Fig. 4). An optimized set of parameters obtained using Nimrod/O significantly improves the dynamical behaviour of the modified Noble 1998 model (solid curve). Importantly, the new set of parameters, which falls within the physiologically acceptable ranges, results in an elimination of the second peak in Ca_i transient (middle panel in Fig. 4). Further, the results demonstrate that Nimrod/O provides a convenient, user-friendly framework for tuning the parameters in the cardiac cell models in an efficient computional manner by taking advantage of parallel batches of evaluations**.** This study provides a valuable platform for future incorporation of the biophysically detailed Ca^{2+} subsystems into whole-cell models of various species.

It is important to note that the use of Nimrod/O highlighted the issues of the parameter sensitivity and over-parameterizations of cardiac ionic models (data not shown). Specifically, the challenges involved in analyzing and characterizing the significance of a given set of parameters in ionic models include (1) potentially fewer-than-necessary constraints being imposed when calculating the objective function, (2) the cardiac ionic models being complex nonlinear systems which have many local minima as opposed to global minima, etc. Parameter estimation in cardiac systems is an ongoing area of research. Thus, while Nimrod/O is a valuable tool in parameter optimization with low computational cost, further studies need to be performed in order to improve the method of finding the optimal set of parameters in a given ventricular single cell model.

4 Conclusions

In this paper we have outlined the steps necessary for updating the ventricular myocytes models with the local Ca^{2+} dynamics. Nimrod/O was used as the tool to incorporate the coupled LCC-RyR models in the place of the existing Ca^{2+} dynamics. To conclude, the incorporation of the local Ca^{2+} dynamics into the Noble 1998 model shows that Nimrod/O is a convenient, user-friendly framework for tuning the parameters in the cardiac cell models in an efficient computational manner by taking advantage of parallel batches of evaluations. Thus, Nimrod/O, provides a valuable, low-computational tool for the incorporation of the biophysically detailed Ca^{2+} subsystems into whole-cell models of various species.

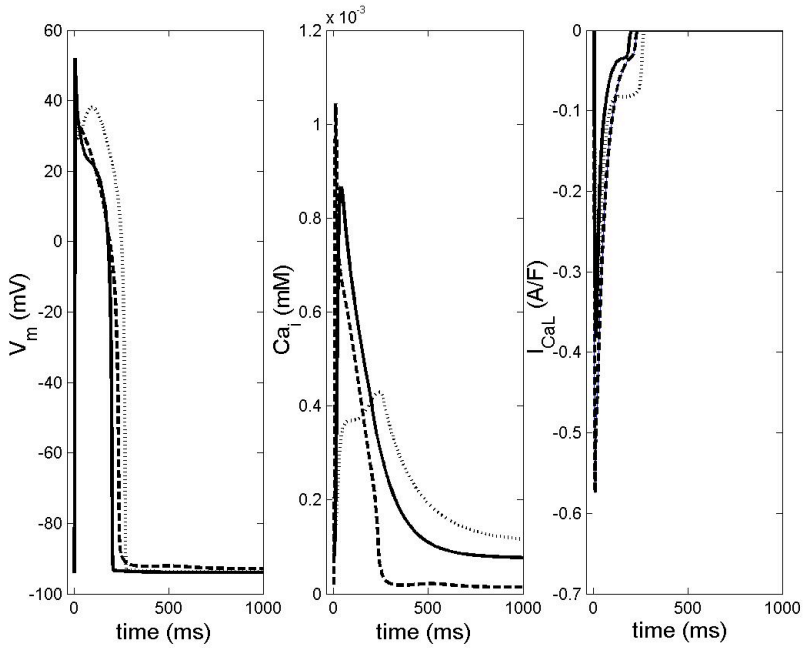


Fig 4. Cardiac model output: The guinea pig Noble 1998 [4] ventricular model modified to include local Ca^{2+} dynamics. Dashed curve represents the Noble 1998 model. Dotted curve shows the modified Noble 1998 model that incorporates the 40-state coupled LCC-RyR Greenstein et al. 2006 model. Solid curve denotes Noble 1998 model modified to include Greenstein et al. 2006 Ca^{2+} dynamics with an optimized set of parameters[2].

Acknowledgements

The cardiac modeling project was supported by EPSRC E–Science Pilot Project in Integrative Biology GR/S72023/01, UK. The Nimrod project has been funded by the Australian Research Council, the Cooperative Research Centre for Enterprise Distributed Systems, the Department of Communications, Information Technology

[2] The optimal set of parameters (corresponding to the solid curve) is as follows: an increase in the maximum rate of the SERCA pump (1.4-fold), an increase in the conductance of RyR (3.5-fold) and LCC (1.5-fold) channels, modified constants of the 10-state LCC Markov model (the transition rate to the Ca^{2+}-dependent inactivation (CDI) state by 1.33-fold; the transition rate out of the CDI-state constant by 0.34-fold; the transition rate out of the closed state by 2.2-fold; the transition state out of the open state by 0.59-fold) and of the 4-state RyR Markov model (the transition rate into the open state from CDI-state 2 by 4.58-fold; the transition rate out of open state into CDI-state 2 by 0.79-fold; the transition rate into the open state from CDI-state 4 by 2.4-fold; the transition rate out of the closed state by 2.1-fold).

and the Arts under a GrangetNet grant, and the Australian Partnership for Advanced Computing.

References

1. Greenstein, J. L., Hinch, R., Winslow, R. L. “Mechanisms of excitation-contraction coupling in an integrative model of the cardiac ventricular myocyte”. Biophys. J. 90, 77-91, (2006)
2. Hinch, R., Greenstein, J. L., Tanskanen, A.J., Xu, L., Winslow, R. L. “A simplified local control model of calcium-induced calcium release in cardiac ventricular myocytes”. Biophys. J. 87, 3723 – 3736, (2004)
3. Noble, D., Varghese, A., Kohl, P., Noble, P. “Improved guinea-pig ventricular cell model incorporating a diadic space, I_{Kr} and I_{ks}, and length- and tension-dependent processes”, Can. J. Cardiol. 14(1), 123-134. (1998)
4. Abramson, D, Lewis A, Peachey T, Fletcher, C., “An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics”, SuperComputing 2001, Denver, Nov (2001)
5. Abramson, D., Sosic R., Giddy J. and Hall B., “Nimrod: A Tool for Performing Parametrised Simulations using Distributed Workstations”, The 4th IEEE Symposium on High Performance Distributed Computing, Virginia, August (1995)
6. Abramson, D, Lewis, A. and Peachy, T., “Nimrod/O: A Tool for Automatic Design Optimization”, The 4th International Conference on Algorithms & Architectures for Parallel Processing (ICA3PP 2000), Hong Kong, 11 - 13 December (2000)
7. Abramson, D., Giddy, J. and Kotler, L. “High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?”, International Parallel and Distributed Processing Symposium (IPDPS), Cancun, Mexico, pp 520- 528, May (2000)
8. Foster, I and Kesselman, C. (editors), The Grid: Blueprint for a New Computing Infrastructure, ISBN: 1558609334, 2nd edition, Morgan Kaufmann, USA, November 18, (2003)
9. Ayyub, S., Abramson, D., Enticott, C., Garic, S., Tan, J. “Executing Large Parameter Sweep Applications on a Multi-VO Testbed”, 7th IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2007, pp 73 – 80, Brazil, (2007)
10. Goscinski, W and Abramson, D. “Legacy Application Deployment over Heterogeneous Grids using Distributed Ant”, IEEE Conference on e-Science and Grid Computing, Melbourne, Dec (2005)

High-throughput protein structure determination using grid computing

Jason W. Schmidberger[1], Blair Bethwaite[4], Colin Enticott[4], Mark A. Bate[1], Steve G. Androulakis[1], Noel Faux[5], Cyril F. Reboul[1,2], Jennifer M. N. Phan[1], James C. Whisstock[1,2], Wojtek J. Goscinski[3] , Slavisa Garic[4], David Abramson[3,4], and Ashley M. Buckle[1,2]

[1]Department of Biochemistry and Molecular Biology,
[2]ARC Centre of Excellence in Structural and Functional Microbial Genomics,
[3]Monash eResearch Centre,
[4]Clayton School of Information Technology,
Monash University, Victoria 3800, Australia.
[1]NICTA Victoria Research Laboratory at The University of Melbourne, Australia.

Abstract

Determining the X-ray crystallographic structures of proteins using the technique of molecular replacement (MR) can be a time and labor-intensive trial-and-error process, involving evaluating tens to hundreds of possible solutions to this complex 3D jigsaw puzzle. For challenging cases indicators of success often do not appear until the later stages of structure refinement, meaning that weeks or even months could be wasted evaluating MR solutions that resist refinement and do not lead to a final structure. In order to improve the chances of success as well as decrease this timeframe, we have developed a novel grid computing approach that performs many MR calculations in parallel, speeding up the process of structure determination from weeks to hours. This high-throughput approach also allows parameter sweeps to be performed in parallel, improving the chances of MR success.

1. Introduction

Proteins perform the functions necessary for life in all organisms. Protein function is to a large extent dictated by the 3-dimensional structure, and thus knowledge of the atomic structure of a protein is a prerequisite to understanding its function. The understanding of protein structure now has a firm role in the molecular basis of all diseases, and as such is a vital underpinning for the future promise of de novo drug design. X-ray crystallography is the most common technique for the structure elucidation of proteins. Briefly, this method involves first the production of large amounts of (usually recombinant) pure protein, followed by crystallization and X-ray diffraction analysis. The atomic structure is then calculated from the diffraction pattern using one of several methods. Over the last 5 years adoption of automation technologies has eased the bottlenecks at the cloning, protein production and crystallization stages. Availability of synchrotron radiation has increased the rate at which high-quality diffraction data can be collected. Although the development of computational methods of structure elucidation has also undergone significant improvement, the high-throughput nature of the pipeline places an increasing emphasis on the computational resources available for structure calculation.

Structure determination can take days to months, and is frequently complicated by the heterogeneity of hardware, software and data

formats encountered. Despite efforts to improve the user-friendliness of software the learning curve for novice structural biologists can be steep, particularly for researchers with a biological sciences background. Furthermore, crystallographers have been slow to harness the power of high-performance distributed computing. In this paper we describe the development of a novel approach to performing common crystallographic calculations in a high throughput fashion, using grid computing.

2. Background

The most common method of crystallographic protein structure determination is molecular replacement (MR). This technique involves using the structure of a protein that shares significant sequence similarity with the protein of unknown structure as a starting point in the structure determination (otherwise known as solving the *phase problem*). The process generally involves four steps: (1) Using sequence-based searching methods such as PSI-BLAST [1] to identify suitable structures that can be used for MR; (2) modification of structures to yield *search models*; (3) Finding the orientation and position of the search model in the unit cell of the target crystal; (4) Refinement of the model.

Molecular replacement has been used to determine the structure of approximately half of the 50,000 structures deposited in the Protein Data Bank (PDB; 67% of 2006 releases were solved by MR [2]). It is anticipated that the proportion determined by MR will grow for three reasons: First, the probability that the unknown target structure belongs to readily identifiable fold is steadily increasing, due to the rapid growth of the PDB. Second, the emergence of more sophisticated sequence searching algorithms, such as profile-profile matching [3], improve the probability of finding a suitable search model, even in cases of very low similarity (<20% identity). Third, MR algorithms consistently improve.

3. Parallel Molecular Replacement

Where the sequence similarity between the unknown target and the search model is high (sequence identity >40%) the success rate of MR is very good, even without optimization of the search model. However, in cases where sequence similarity is low (identity <30%) MR, and subsequent structure refinement becomes non-trivial, and emphasis must be placed on the optimization of the search model. A key breakthrough in successfully applying the MR approach to situations where sequence identity is low was the development of the PHASER maximum likelihood approach [4].

Even in cases where a MR solution with low overall sequence identity can be obtained these solutions are commonly challenging to refine (the so called “model bias” trap). This situation occurs where errors in regions of the starting model cannot be adequately identified and corrected due to model bias. There are several criteria that affect the outcome of the MR calculation; 1) structural similarity between search model and target structure (measured by root mean square deviation (RMSD)); 2) percentage of residues missing from the search model (*coverage*); 3) the amount of conserved side chains that are expected to remain structurally conserved (for example in the protein interior). These factors, and thus the outcome of the MR calculation, can be influenced by improvement of the search model, using several methods. The simplest approach is to remove regions of the structure that are predicted to be different in the search model and target, typically loops. However, this process is a subjective one and relies on

sequence alignments, which are often incorrect, particularly at low sequence identity. Thus it is often unclear which loops should be removed and how much of the loop should be removed, and each model must be tested. We have developed a robust solution to this method, called *sieving*, which produces

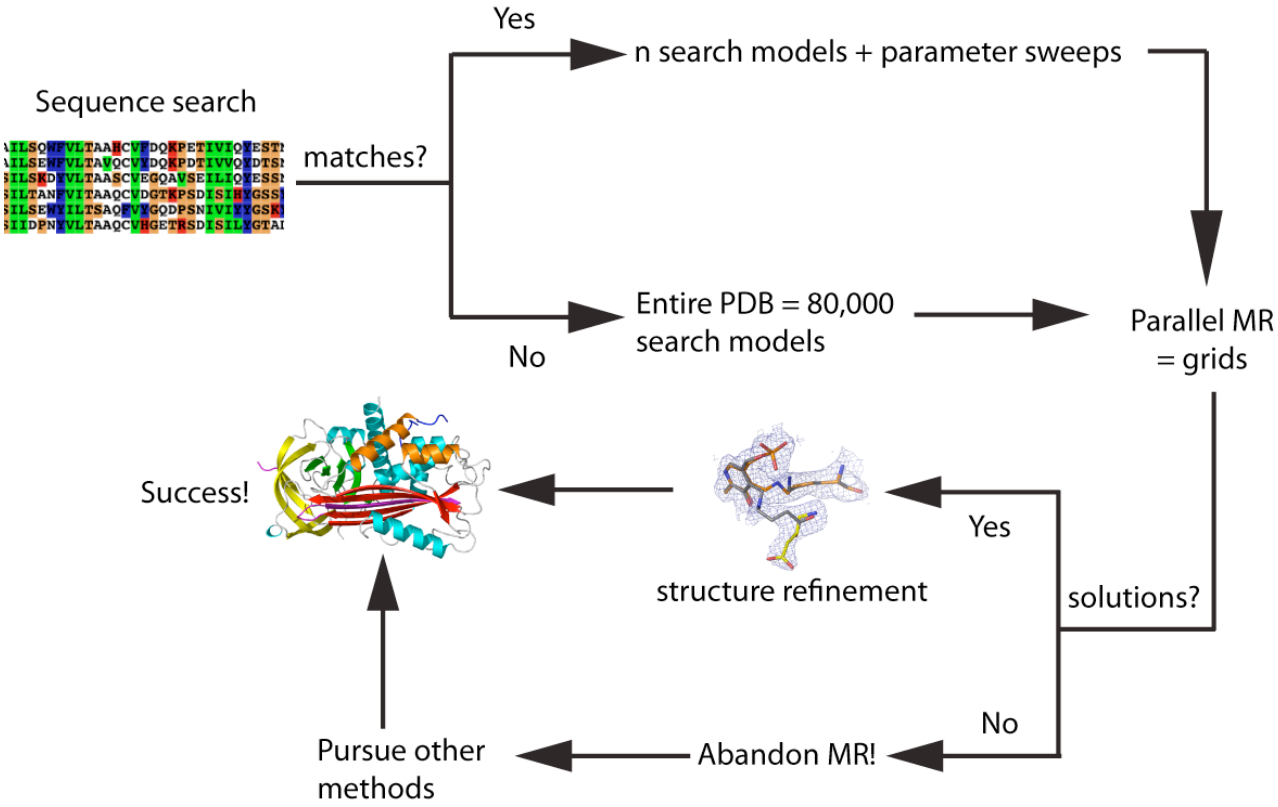


Figure 1 – Parallel MR Approach

search models with structurally divergent regions removed in an objective fashion (Schmidberger *et al.,* unpublished data). The ideal starting model (*e.g.* one with least model bias) is difficult to obtain *a priori,* however it is possible to test multiple sieved models and asses the refinement process using statistically robust validation, providing a generally applicable method for model bias reduction. A further method of search model improvement leverages protein flexibility. Proteins are typically not rigid but undergo conformational changes, resulting in a population of many distinct conformational states. This can be modelled using the technique of Normal Mode Analysis (NMA [5]), which produces many alternative molecular conformations that can all be tested as search models in the MR calculation. In some cases the true symmetry of the crystal is unknown and several alternatives must be tested in the MR calculation. Finally, the estimated RMSD between the search model and unknown structure can affect the outcome of the MR calculation, leading in the worst case to probable solutions being missed.

Therefore, for challenging cases the combination of multiple search models produced by sieving and NMA, symmetry and RMSD values makes MR potentially time and labor intensive, and puts an emphasis on the availability and power of computational resources.

Given the complexities of the MR technique outlined above we have developed a novel grid computing approach that is able to perform independent MR calculations using hundreds to thousands of candidate search

models in parallel, whilst also performing near-exhaustive parameter-sweeps in order to increase chances of success. Furthermore, we have extended this approach so that the entire protein fold universe (currently ~80,000 folds) can be tested. This massively parallel approach may be useful in cases where the similarity between a protein of unknown structure and a "known fold" cannot be detected by sequence matching methods, despite them sharing the same fold. For such proteins, an MR-based approach may be achievable, but up until recently, the computational resources required for such an approach would be prohibitive. However, the exponential growth of computing power and recent advances in harnessing this power in a massively parallel fashion, using grid computing, means this approach is now feasible. Figure 1 summarizes the rationale behind this approach.

4. Implementation

We have implemented hierarchical grid-based approach that leverages a range of distributed computational resources. Generally, the approach performs multiple PHASER-associated MR calculations across a grid of networked computers, permitting high-throughput MR. This approach is summarized in Figure 2.

There is significant heterogeneity in the resources available to us accessed by a range of different middleware solutions. For example, a collection of Apple Macintosh's in the department of biochemistry at Monash University use Apple's proprietary Xgrid technology. We have exploited the rapid and easy implementation of Xgrid for developing the parallel MR methodology and for proof-of principle testing. Once convinced of its merit and for larger scale computations, we used a combination of a large Condor Pool, that aggregates many of the student laboratory machines, and a range of clusters distributed globally. These latter resources were accessed via Globus and the Nimrod/G middleware. These are discussed in more detail below.

4.1 Xgrid

We have built a web based application written in Java/JSP and Ruby, and taking advantage of Apple Xgrid technology (see: http://www.apple.com/server/macosx/technology/xgrid.html), which we call *MR Grid*. Designed to interface with a user defined Xgrid resource the package manages the distribution of multiple MR runs to the available nodes on the grid and reports all returned results. Utilizing the maximum likelihood based molecular replacement program PHASER [4], MR Grid enables the user to retrieve and manage the results of hundreds of MR calculations via a single web interface, as well as broadening the range of strategies that can be attempted, increasing the likelihood of success.

MR Grid is distributed as a self-contained software package, and downloaded and executed across a local grid resource. Once set up MR Grid is accessed through a web portal. Apple Xgrid software is preinstalled on Apple operating systems OS X 10.4 and 10.5, allowing machines to be

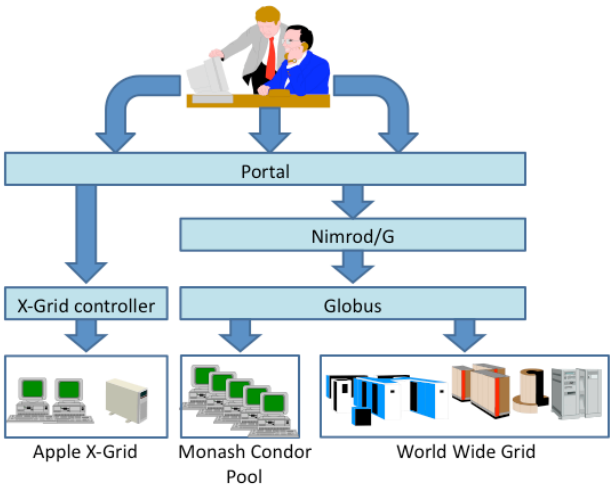


Figure 2 – Parallel MR grid architecture

configured as Xgrid clients by simply activating a setting in system preferences. By default MR Grid processes on the client are given low priority, such that the client remains fully responsive. The remaining requirement is a networked machine acting as the Xgrid controller. This can be running either the server or client version of Mac OS 10.4/10.5.

MR Grid takes as input the X-ray diffraction data and a compressed file of search models. MR Grid then parses the input and distributes jobs to available nodes on an available grid resource. Each job runs to completion independent of all other jobs, and a URL where the results of the submission can be accessed is returned to the user. Job distribution and queuing is performed entirely by the Xgrid controller and requires no programming. MR Grid source can be found at http://code.google.com/p/mrgrid/.

4.2. Condor and World Wide Grid

MR Grid is well suited to performing parallel MR on a modest laboratory-based network of Apple computers, and performs useful validation of our approach. In order to attack more challenging problems, we have leveraged two different classes of resource that were available to us, namely a ~1000 CPU Condor pool built from otherwise idle desktop machines in Monash teaching laboratories; and a World Wide Grid of machines leveraging computers (mostly clusters or Condor pools) in an Australian University Enterprise Grid, the Pacific Rim and Grid Middleware Assembly (PRAGMA) testbed, and the US based Open Science Grid (OSG). This provides thousands of processors and has allowed us to perform high throughput MR based calculations.

We have used Nimrod/G [6] to allow the distribution and execution of a large number of jobs to both resources. Nimrod/G interfaces with the Globus middleware that provides a common access layer for all resources. Thus, we are able to access the Monash Condor Pool in exactly the same way as the Grid resources that are available through our Global collaborations. Combining all of these machines, has allowed us to perform an extremely large MR experiment consisting of some 80,000 PHASER executions.

Table 1. List of test case proteins, extracted from the Protein DataBank (PDB). Details about respective datasets are also listed

|PDB id	Protein Name/Type	Space Group	Resolution Limit (Å)	Molecular Mass (Da)	Ave. %ID	# Search Models	# SGs (or RMSDs) tested	# Jobs (RMSD)	Grid Run time (hh:mm:ss)	Linear Run time (hh:mm:ss)
2GPZ	transthyretin-like protein	$P6$	2.5	12700	66.4	4	6 (5)	24 (20)	0:13:24 (0:15:05)	1:16:53 (0:57:33)
2NO4	Haloacid Dehalogenase	$P3_121$	1.9	24000	37.7	5	3	15	7:16:48	22:18:36
2CWQ	Hypothetical protein TTHA0727	$P3_121$	1.9	12581	<30	8	3	24	5:09:03	23:54:00
2ENX	Mn-dependant inorganic pyrophosphatase	$H32$	2.8	33597	57.5	7	1	7	0:04:07	0:17:39
2RH5	Adenylate kinase	$C222_1$	2.48	23231	43.2	8	2	16	0:20:36	0:40:38
1S3G	Adenylate kinase	$P3_121$	2.25	23888	41.3	8	3 (4)	24 (32)	0:25:12 (0:33:34)	1:38:19 (5:08:41)
2JCB	5-Formyl-tetrahydrofolate cycloligase	$P1$	1.6	23385	31	4	1 (4)	4 (16)	1:06:58 (1:21:00)	7:18:00 (6:55:35)
2H74	Thioredoxin	$P6_1$	2.4	11807	49.5	9	6	54	0:22:12	4:32:28
1FB0	Thioredoxin	$P3_121$	2.26	11782	45.1	9	3 (5)	27 (45)	0:29:40 (0:12:28)	3:24:34 (1:44:54)
2MM1	Myoglobin	$P3_221$	2.8	17184	53.9	12	3	36	0:10:42	1:20:41

5. Experiments and Results

In all cases MR calculations using the program PHASER produce possible solutions having a maximum likelihood Z-score. Z-scores greater than 7 were chosen as probable solutions worthy of structure refinement. Specific methodologies and test data were chosen based upon the class of grid resource available, and are described in detail below.

5.1 Xgrid

A set of 10 proteins were used as test cases, representing 8 different SCOP [7] families (Table 1), and allowing for the parallel execution of 4 to 54 jobs at any one time. PDB entries were selected on the basis of having 3 or more homologous structures in the PDB, with datasets from a range of crystal symmetries. MR search models were generally chosen on the basis of a >30% sequence identity across >75% of the monomer of interest (*i.e.* no partial matches).

Experimental data taken from PDB for the 10 proteins listed in Table 1 were each used in test case experiments in order to demonstrate the utility of the system under typical situations. For each protein example, data were screened against each homologue search model (including self), searching all possible space group symmetries. In this way, the number of jobs submitted to our local grid varied between 4 and 54, and the corresponding speed up factors showed a clear linear relationship. Featuring an average speed up value of 5.7 across all the tests, it is clear that our approach has the capacity to significantly reduce the time taken to achieve a MR result when screening numerous parameters, thus demonstrating the validity of the approach.

5.2 Condor

Having demonstrated the benefits of a parallel approach to MR, we sought to apply it to a challenging, real-world case. We chose a specific X-ray dataset in our laboratory that had proved recalcitrant to several methods of structure determination, including MR (unpublished). The target protein was of molecular weight of 44500 Da, with one molecule in the crystal asymmetric unit (~53% solvent). Using sequence searching we found two potential search models in the PDB, having 24 and 16% sequence identity, respectively. X-ray diffraction data extended to 1.6Å resolution. For such a challenging case we chose to perform extensive parameter sweeps coupled with testing alternative models from a Normal Mode Analysis calculation (using the *El Nemo* server: http://www.igs.cnrs-mrs.fr/elnemo/) as well as a sieving approach (Schmidberger *et al,* unpublished). NMA analysis generated 11 models for each of the 5 lowest modes, giving 55 models in total. Each mode was then sieved separately, giving 495 models in total [11 x 9 (sieve levels) x 5 (NMA modes)]. When combined with RMSD parameter sweeps (5 alternatives) this produced a total of 2475 MR runs.

Calculations executed on up to 469 Condor nodes at once, and completed within 15 hours. After sorting the results according to the maximum likelihood Z-score in PHASER, we obtained a unique solution having a Z score of 7.5 (previous manual MR calculations failed to produce a Z-score greater than 5.5). After rigid-body refinement using the program REFMAC [8] the model was subjected to automatic model building and structure refinement using the program ARP/wARP [9]. This resulted in a near-complete structure (329 of 390 residues (85%) built), having crystallographic R(work) and R(free) values of 0.20 and 0.22, respectively (R-values are the typical measure of model

correctness; typically correct, fully refined models have R(free) < 0.30).

5.3 World Wide Grid

In order to test the validity of the ambitious approach of using the entire set of known protein structures in a MR calculation, we chose two test cases of contrasting complexity. The first case, *Thioredoxin* represents a typical example of a small-sized protein: (Molecular weight 11.6 kDa; PDB ID 2E0Q, one molecule per crystal asymmetric unit, data resolution 1.5 Å). We generated approx 70,988 search models from the PDB using the domain classification according to the SCOP database [7]. The side chains of all residues were truncated to alanine, and all loops were removed prior to the MR calculation. We performed 70988 independent MR calculations using the Thioredoxin test dataset across a total of 300 nodes worldwide. The entire run took 94 hours to complete, producing high Z-scores, as expected, for search models having clear structural homology with thioredoxin.

For the second test we selected a more challenging example of a larger protein; *Thiamin Phosphate Synthase* (Molecular weight 25 kDa; PDB ID 2TPS, two molecules per crystal asymmetric unit, data resolution 1.25 Å). This MR calculation produced 70,988 independent MR runs, searching for one molecule in the crystal asymmetric unit, and a total of 1050 nodes worldwide, and took 52 hours to complete. Nodes utilized in both runs included resources at VPAC and Monash University in Victoria, Australia and PRAGMA grid resources in Japan, Switzerland, Thailand and the United States. In total this experiment utilized approximately half a million CPU hours.

6. Conclusions and Future Plans

Our MR calculations over small laboratory-based networks of Apple computers show clearly how tens to hundreds of search models can be performed in parallel, along with parameter sweeps, increasing the chances of success in a relatively short timeframe. Testing this approach using a challenging, unsolved X-ray dataset required a larger grid, and we were able to show that for our real-world case, parallel MR calculations could be performed on a medium-sized university campus grid in less than 15 hours (e.g., overnight). Importantly, this experiment allowed us to perform near-exhaustive parameter sweeps, and resulted in a successful structure determination of a protein structure that had previously resisted structure determination by conventional (e.g., serial) MR calculations. This result shows clearly the promise of performing MR in a parallel fashion. By extending this concept to cases where no suitable search models could be obtained, we demonstrated that using worldwide grid resources available to academic scientists, MR could be performed using the entire known protein structure universe as independent search models, in a timeframe of two days in the case of a medium sized protein.

We expect that the current trend towards multi-core architecture will strengthen our parallel MR approach. It is important to note that apart from the clear advantage of rapid structure determination, even negative results (e.g., failure to find MR solutions) will prove useful: an unsuccessful exhaustively-parallel MR calculation allows the protein crystallographer to make an objective decision on alternative methods to be pursued, at the early stages of the project, offering potential labor and cost savings.

7. Acknowledgements

We thank Ruby Law, Michelle Dunstone, and Randy Read for helpful discussions. We also

thank our colleagues in PRAGMA, the Enterprise Grid, The Swiss National Grid and the Open Science Grid for their assistance. We acknowledge the resources provided by the Open Science Grid, which is supported by the National Science Foundation and the U.S. Department of Energy's Office of Science; the resources provided by the Pacific Rim Applications and Grid Middleware Assembly and its member organizations, PRAGMA member institutes are funded by funding agencies of their respective governments. PRAGMA is supported by NSF Grant No. INT-0216895, INT-0314015 and OCI-0627026. We acknowledge support from the DEST (DIISR) funded DART and ARCHER projects, support from the Australian Research Council (ARC), the National Health and Medical Research Council (NHMRC), Victorian Partnership for Advanced Computing, the Victorian Bioinformatics Consortium, the state government of Victoria (Australia), the Monash library and the Monash e-Research Centre. A. M. B is an NHMRC Senior Research Fellow. J. C. W. is an ARC Federation Fellow, a Monash University Logan Fellow and an honorary NHMRC Principal Research Fellow. D. A. is an ARC Professorial Research Fellow.

8. References

[1] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman, Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res 25 (1997) 3389-402.

[2] F. Long, A.A. Vagin, P. Young, and G.N. Murshudov, BALBES: a molecular-replacement pipeline. Acta Crystallogr D Biol Crystallogr 64 (2008) 125-32.

[3] L. Jaroszewski, L. Rychlewski, Z. Li, W. Li, and A. Godzik, FFAS03: a server for profile--profile sequence alignments. Nucleic Acids Res 33 (2005) W284-8.

[4] A.J. McCoy, R.W. Grosse-Kunstleve, P.D. Adams, M.D. Winn, L.C. Storoni, and R.J. Read, Phaser crystallographic software. Journal of Applied Crystallography 40 (2007) 658-674.

[5] K. Suhre, and Y.H. Sanejouand, ElNemo: a normal mode web server for protein movement analysis and the generation of templates for molecular replacement. Nucleic Acids Res 32 (2004) W610-4.

[6] D. Abramson, Giddy, J. and Kotler, L., High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?, International Parallel and Distributed Processing Symposium (IPDPS), Cancun, Mexico, 2000, pp. 520- 528.

[7] A.G. Murzin, S.E. Brenner, T. Hubbard, and C. Chothia, SCOP: a structural classification of proteins database for the investigation of sequences and structures. J Mol Biol 247 (1995) 536-40.

[8] G.N. Murshudov, A.A. Vagin, and E.J. Dodson, Refinement of macromolecular structures by the maximum-likelihood method. Acta Crystallogr D Biol Crystallogr 53 (1997) 240-55.

[9] S.X. Cohen, R.J. Morris, F.J. Fernandez, M. Ben Jelloul, M. Kakaris, V. Parthasarathy, V.S. Lamzin, G.J. Kleywegt, and A. Perrakis, Towards complete validated models in the next generation of ARP/wARP. Acta Crystallogr D Biol Crystallogr 60 (2004) 2222-9.

Chapter 9: Grid Testbeds

There has been a great deal of work in building the low level infrastructure required for the Grid. In this chapter I include two papers that contribute to this debate. (38) and (39) discusses my work in the Pacific Rim Applications and Grid Middleware Assembly (PRAGMA), a group of researchers from many institutions across the pacific Rim. Together we built a real production grid called PRAGMA Grid,, and used this to solve applications in science and engineering. (38) discusses the technology under the case study presented in paper (34). (39) talks about Grid interoperability between PRAGMA Grid and other infrastructure.

(1) Abramson, D., Lynch, A., Takemiya, H., Tanimura, Y., Date, S., Nakamura, H., Jeong, K., Zhu, J., Lu, Z., Lee, H., Wang, C., Shih, H., Molina, T., Baldridge, K., Li, W. and Arzberger, P. "Deploying Scientific Applications to the PRAGMA Grid Testbed: Strategies and Lessons", 6th IEEE International Symposium on Cluster Computing and the Grid 16-19 May 2006, Singapore. 9-1 .. 9-8

(2) Bethwaite, B, Abramson, D and Buckle, A. "Grid Interoperability: An Experiment in Bridging Grid Islands", PRAGMA Workshop at e-Science, 2008, 4th IEEE International Conference on e-Science, Indiana, Dec 8th – 12th, 2008. 9-9 .. 9-16

Contributions

There are a large number of authors on paper (40) from the PRAGMA community. This effort was led by myself and Wilfred Li from UCSD; Lynch, from Monash, provided the case study in climate studies. Paper (41) was led by Bethwaite, a research associate under my supervision; Buckle provided the case study in structural biology.

Deploying Scientific Applications to the PRAGMA Grid Testbed: Strategies and Lessons

David Abramson, Amanda Lynch, Hiroshi Takemiya, Yusuke Tanimura, Susumu Date, Haruki Nakamura, Karpjoo Jeong, Suntae Hwang, Ji Zhu, Zhong-hua Lu, Celine Amoreira, Kim Baldridge, Hurng-Chun Lee, Chi-Wei Wang, Horng-Liang Shih, Tomas Molina, Wilfred W. Li, Peter A. Arzberger

***Abstract* — Recent advances in grid infrastructure and middleware development have enabled various types of applications in science and engineering to be deployed on the grid. The characteristics of these applications and the diverse infrastructure and middleware solutions developed, utilized or adapted by PRAGMA member institutes are summarized. The applications include those for climate modeling, computational chemistry, bioinformatics and computational genomics, remote control of instruments, and distributed databases. Many of the applications are deployed to the PRAGMA grid testbed in routine basis experiments. Strategies for deploying applications without modifications, and those taking advantage of new programming models on the grid are explored and valuable lessons learned are reported. Comprehensive end to end solutions from PRAGMA member institutes that provide important grid middleware components and generalized models of integrating applications and instruments on the grid are also described.**

***Index Terms*—PRAGMA, grid, applications, deployment, strategies, lessons**

I. Introduction

The grid environment has evolved rapidly over the past few years, with a number of scientific applications serving as drivers for development of middleware and grid infrastructure [1]. While the exponential growth in processor power, storage, bandwidth and fiber lays the foundation for the new computing infrastructure, the grid will only succeed if it attracts users, and meets their needs [2]. Scientists and engineers often are the first to plunge into the new technology and come out with valuable lessons learned for posterity.

Over the past 10 years, development effort in grid middleware in projects such as Globus [3] has enabled a middleware layer that handles essential functionalities such as user authentication, authorization, data storage and transfer. However, in the last 3 years, a new layer of tools have emerged or matured that build on top the "lower middleware" and further enhance the ease of use for scientific and engineering applications (Figure 1).

In this paper, we describe applications that are deployed on the PRAGMA testbed [4] with or without modifications. We also describe some comprehensive grid application environments that are developed by PRAGMA member institutes that provide end to end solutions for grid computing.

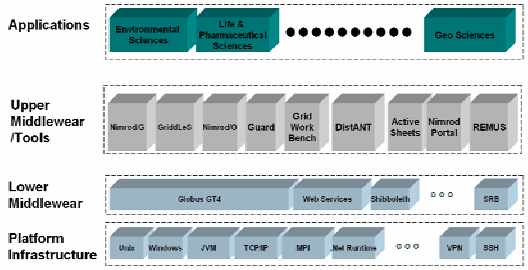


Figure 1. User applications and experiences on the grid are enhanced by upper middleware and tools

II. Deployment Strategies

A. Applications without code modifications

Parameter sweep applications are those that iterate over certain sets of parameters for a particular application as required by the research problem at hand. Here we'll discuss the climate modelling and computational chemistry complex workflow, as well as bioinformatics applications using Nimrod [5] and Gfarm [6]. These middleware support the deployment of applications without requiring changes in existing applications. The no change or minimal changes on "legacy" applications significantly increases the appeal of the grid to a wider audience.

1) Paeloclimate experiments with Nimrod

The Australian monsoon is critical to the environment and

Manuscript received November 30, 2005. The author names are sorted in alphabetical order by name of the country, institute, and last name where appropriate. PRAGMA member institutes are funded by funding agencies of their respective governments. PRAGMA is supported by NSF Grant No. INT-0216895 and INT-0314015. WWL, PAA also wishes to acknowledge NIH NCRR P41 RR08605 to NBCR. WWL wishes to acknowledge Fang-Pang Lin, Habibah Wahab, and the participants at the 1st South Asia International Joint Research and Training workshop hosted by NCHC, Taiwan, for their valuable input.

D. Abramson and A. Lynch are with the University University of Monash, Australia, Victoria 3800, Australia. (e-mail: {david.abramson, amanda.lynch}@infotech.monash.edu.au). H. Takemiya and Y. Tanimura are with AIST, Japan (e-mail: {h-takemiya, yusuke.tanimura}@aist.go.jp). S. Date and H. Nakamura are with CMC, and Institute for Protein Research at Osaka University, Japan, repectively. ({date@ais.cmc,nakamura@protein} .osaka-u.ac.jp). K. Jeong and S. Hwang are with {Konkuk, Kookmin} University Korea. (e-mail: {jeongk@konkuk, sthwang@kookmin}.ac.kr). J. Zhu and Z. Lu are with CNIC, CAS, China. (e-mail: {zhuji, zhlu}@sccas.cn). C. Amoreira and K.Baldridge are with University of Zurich, Switzerland (email {amoreira,kimb}@oci.unizh.ch). H. Lee, C. Wang, H. Shih are with Academia Sinica Grid Computing, Taiwan. (e-mail: {hclee,chiwei,hlshih@ccweb.sinica.edu.tw). T. Molina, K. Baldridge, W. W. Li, and P. A. Arzberger are with University of California, San Diego, La Jolla 92093, USA. (phone: 858-822-0974; fax: 858-822-0861; e-mail: molina@ncmir.ucsd.edu; *wilfred@sdsc.edu*; *parzberg@ucsd.edu*).

economy of the north Australian region, which produces wealth for Australia out of proportion to its population. It delivers life-sustaining moisture to a dry continent, although lightning associated also causes devastating fires. During the past, the monsoon has varied both spatially and in intensity. Contemporary ecosystems have adapted to these extremes, but how tightly linked are burning, vegetation, and rainfall? What might these linkages mean for future Australian water resources?

Interactions between atmosphere, ocean and land in the context of the Australian monsoon are complex, as they result from feedbacks that operate on a variety of spatial and temporal scales. Until now, computational and data volume limitations have hindered efforts to reach a full understanding of the biophysical processes and the mechanisms for long term variation in the natural monsoon system. Performing simulations of the type and extent needed to understand monsoon dynamics require more computational resources than one is likely to obtain from any single high performance computing centre (HPC) in Australia. Moreover, the databases required to drive the simulations are distributed and replicated at different centers. As a result, there are "few secure facts concerning when and why the Australian summer monsoon developed or how it has varied" [7]. In this context, an ongoing initiative seeks to develop the capability for the simulation and analysis of the natural variability of the Australian monsoon.

-- Use of Nimrod: Over the past 10 years, Abramson *et al* have developed a software tool called Nimrod/G, which allows a user to migrate a particular class of applications to the Grid [5, 8]. Specifically, it automates the execution of parameter sweep and search applications (parameter studies) over global computational grids. Nimrod is particularly novel because it supports user-defined deadline and budget constraints for scheduling computations and manages the supply and demand of resources in the Grid using an experimental computational economy. Nimrod/G supports the type of parameter study required in the paeloclimate modeling experiment to understand the Australian monsoon. It allows the scientists to vary the initial conditions and various other parameters of the climate models, as well as performing a large number of monte-carlo style simulations.

-- Pilot Study over PRAGMA testbed: The pilot study utilizes an existing Australian atmosphere-land model, the Cubic Conformal Atmospheric Model (C-CAM) developed at the CSIRO Division of Atmospheric Research. C-CAM has 18 vertical levels, hydrostatic dynamics, and a single-layer canopy with 44 vegetation types. The sea surface temperatures are prescribed as well as the wind components using a far field nudging technique. This configuration reproduces the Australian Monsoon climatologically in a reasonable way.

A one month simulation using the single CPU version of C-CAM takes 150 minutes to run on a 2.8 GHz INTEL XEON architecture using the INTEL Fortran 95 v8.0 compiler with modest optimization enabled. Typical paleoclimate experiments are on the order of 30 years for a single realization – such an experiment then requires 38 days of wall clock time and 100 GB of forcing data to complete. However, because the earth's climate is a chaotic system with thousands, if not millions, of degrees of freedom, an appropriate experimental design requires ensembles of realizations and ranges of sensitivity experiments. These may be considered to be highly parallel in nature even when the parallelized version of C-CAM is not used.

The core component for process control and the distribution of the runs is Nimrod/G, and is described in more detail in an accompanying paper [4]. The use of nimrod/G enabled successful executions of the C-CAM on the PRAGMA testbed.

2) QM/APBS in computational chemistry

Rational drug design relies on computational models that determine whether and how small drug ligands interact with large molecules like proteins. This is done by calculating the ligand-protein configuration that minimizes the binding energy. The methods are often complex because the code performs both energy calculations as well as nonlinear optimizations. Baldridge *et al* has proposed an alternative framework, enabling user-specific exploration using a QM (quantum mechanics)/APBS (adaptive Poisson-Boltzmann Solver)/MD (molecular dynamics) hybrid method. The ligand is treated using QM methods, and the full complex is treated using classical electrostatics (APBS) [9] and empirical force field (MD) methodology. The first phase involves only QM/APBS, with the ultimate goal being to envelop the classical electrostatics directly into the QM procedure, and combine with MD methods.

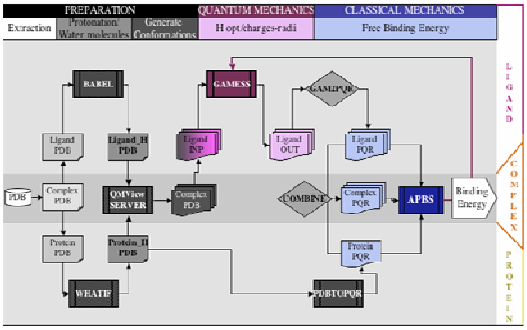


Figure 2. Various steps involved in the hybrid method for study of ligand-protein interactions.

Computationally, the hybrid methodology involves many individual steps, many of which we have now eliminated through creative middleware and web services implementations, most noticeably on the PRAGMA testbed, through the use of the Nimrod/G. The electrostatic effects are numerically solved using APBS. The atomic charges and radii for the ligand are determined using QM. Additionally, QM provide more accurate structural and energetic information for the ligand, an important factor in the determination of which structure is preferred in the pocket of the protein or enzyme. All charges and radii must be specified for the electrostatics computation.

The overall goal is to understand the binding energy and mechanism associated with the complex formation of the

ligand and protein, and how they vary with position, substitution (residue mutation), and environmental conditions. In particular, it is desirable to define a large set of parameters required in defining the binding energy, parameters having to do with ligand positioning and environmental conditions, with the added flexibility to run investigations over a wide set of possible mutations in structure. In addition, it is also important to determine more accurate energy functions for predicting the total binding energy for a protein-ligand complex. The total theoretical method involves multiple tools and resources, including molecular modeling software, databases, and auxiliary tools, all managed by the grid middleware tool, Nimrod, discussed above.

3) Protein annotation studies using iGAP

The international genome sequencing effort has steadily produced a large number of complete genomes. Since 1995, more than 180 complete, and over 1000 partial proteomes have been made publicly available. Progress is being made towards high quality proteome annotation through a combination of high throughput computation as well as manual curation. Increasingly the new knowledge is translated into tangible diagnostic and remedial procedures for the benefits of public health and education. Grid technology promises to meet the increasing demand in large scale computation and simulation in the fields of computational biology, chemistry and bioinformatics. The integrative Genome Annotation Pipeline, iGAP [10], provides functional annotation using known protein structural information. The computational requirement of iGAP and the initial experience in using AppLeS Parameter Sweep Template (APST) [11] to deploy it on the grid has been previously described by Li *et al* [12].

While the previous system provides many rich features and works well for the dedicated workflow, it requires detailed knowledge to operate and requires significant effort to generalize to other applications. Ease of use is fundamental for the widespread adoption of grid technologies in life sciences, where application scientists do not have the time to keep up with the ever changing grid computation standards and models. As Moore's law continues to hold true, commodity computing clusters is becoming a reality on university campuses across the globe. While this trend is fundamental to the maturation of the grid, it also poses a new challenge. Many users prefer to run bioinformatics applications within a cluster environment, which is the most reliable production environment to date, despite recent advances in grid middleware technology.

One problem often experienced in both the cluster and grid environment is the limitation of file I/O. In a cluster environment, if an application generates a lot of intermediary and output files, the load on the NFS server may become quite high, as the number of compute nodes increases, and I/O becomes a rate limiting step. Code modification is required to move the file I/O to local disks on compute nodes. Additional code is required to reliably transfer the end results from the compute nodes back to the NFS server. Even with a dedicated cluster where local disk may be used for data storage, it becomes difficult to find the results on various compute nodes if one wishes to revisit the data after a certain period of time.

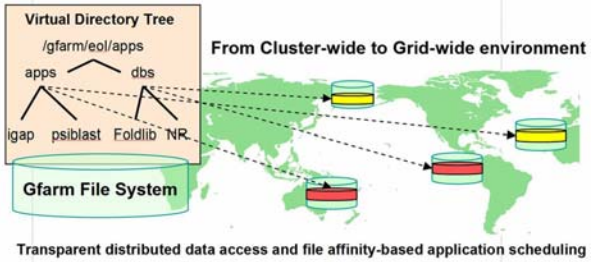


Figure 3. Gfarm file system enables centralized distribution of biological databases and applications.

In a grid environment, network latency, bandwidth shortage, and overhead in file transfer are often prohibitive to the effective grid deployment of legacy applications which produce many output files with sizes ranging from tens of kilobytes to tens of megabytes. Gfarm™ provides a scalable and transparent solution, which enables effective use of not only the distributed computing power, but also the disk storage space, through a familiar virtual file system view (Figure 3) [4, 6]. The capability of Gfarm to support "legacy" applications without code modification has proven valuable for lowering the cost of entry to the grid. Additional metascheduling using the Community Scheduler Framework 4 (CSF4) [13] provides the cross site scheduling abilities offered by Nimrod/G or APST, with additional features being developed for a full fledge metascheduler.

B. Applications with code modifications

While running applications without code modifications allows great flexibility and shields the application scientists from the overhead of learning about grid technology and programming models, new programming models may enhance the performance and fault tolerance of applications re/designed for the grid. These cases are explored below using bioinformatics applications, and QM/MD/MM (molecular mechanics) studies.

1) Building high throughput BLAST service using MPICH-G2

Following the successes in genomic sequence analysis, BLAST has been widely used as a fundamental tool in majority of bioinformatics applications. Thus the efficiency of BLAST becomes an essential metric of application performance. mpiBLAST, developed by LANL, is a parallel BLAST implemented by using the NCBI toolbox libraries. The database splitting scheme improves the BLAST performance without requiring high-end computers [14].

From the service perspective, the fast growing database and user demands of BLAST searches require resources that cannot be offered by traditional PC clusters. Considering the limited resources in computing center and the fact that several life science institutes are running BLAST on their own computing resources, the idea of building high throughput BLAST services is to link these resources as a single computing pool by using the Grid technology so that one can leverage the idle computing slots to facilitate on time

consuming and on-demand BLAST searches.

-- Modifications to mpiBLAST: Lee *et al* aims to enable cross-cluster CPU and database sharing features in mpiBLAST on the GT2 based PRAGMA testbed. Figure 4 shows the schematic diagram as well as the workflow of the GT2-enabled mpiBLAST. Porting mpi- BLAST on PRAGMA testbed can be done simply by re-compiling the source code with the GT2-enabled MPI implementation, MPICH-g2. With the use of GT2 DUROC (Dynamic-Update Request Online Coallocator), mpiBLAST-g2 can handle MPI processes running in different clusters; however, the job request needs to be consistent with the mpiBLAST-g2 configuration at each site. Instead of using mpirun, a mpiBLAST-g2 job submitter was developed to produce a site-dependent RSL.

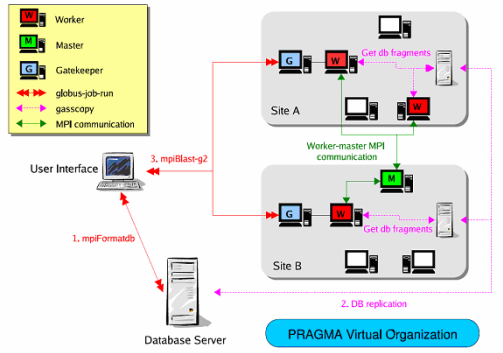


Figure 4. Using MPICH-G2 for MPI-BLAST

To reduce the I/O overhead of BLAST search, simple copy is used in mpiBLAST to fetch the database fragments from a central database repository to local disk before the searching. This limits the location of the repository to an internal network in order to minimize security risks. In mpiBLAST-g2, the copy method was replaced by GASS (Global Access to Secondary Storage) to adopt the GSI (Grid Security Infrastructure) based data access protocol for global database sharing.

-- Observations: The test runs on PRAGMA testbed show that mpiBLAST-g2 can leverage the idle computing resources from multiple clusters to improve the throughput of BLAST service; however, the MPI job in cross-cluster mode has two major issues:

(1) Inbound connectivity is required on each CPU. In the PRAGMA testbed, not all resources can offer public IPs for cross-cluster MPI applications due to site administration policies. Nevertheless, resources without inbound connectivity can still be used to run small jobs in single-cluster mode. For instance, the mpiBLAST-g2 job submitter can select resources in an intelligent way with the use of a static site information table and switch jobs in between single and multiple cluster modes based on the characteristics of the selected resources.

(2) Job hangs due to single CPU failure. Since the grid is a dynamic environment, some unexpected transient network problem might result in a single CPU failure thus causing the whole MPI job to hang while waiting for a response from the failed process. This can lead to inefficient use of grid resources. Although one can periodically check and cleanup the unhealthy jobs in the backend, a better way to address this issue is to introduce a more resilient framework to handle distributed jobs on the grid. Several possibilities, such as fault tolerance MPI implementations and master-worker framework like DIANE [15], are under investigation.

2) QM/MD simulations in material science & engineering

The hybrid QM/MD technique for atomistic simulations is becoming more and more important in the field of modern material science and engineering, such as designing future electronic devices or micro-machines. In order to design delicate and robust devices, designers need a microscopic analysis such as the stress distribution of the material or the deformation process. The hybrid QM/MD simulation [16] is a combination of a classical MD simulation with a Density Functional Theory (DFT)-based QM simulation. The simulation calculates the behavior of atoms in the entire region based on the MD scheme, while QM simulation is performed in interesting regions to improve the MD result.

Typically this type of simulation is difficult to execute on a single cluster, because thousands of CPUs must be used over several months to calculate the result on the real problem even using the hybrid method. Takemiya *et al* redesigned the QM/MD simulation code to use ninf-G for grid deployment and job scheduling using GridRPC.

a) QM/MD simulation code redesign considerations

In implementing the code, the following three requirements are deemed to be very important when executing a large scale simulation on the grid for a long time.

(1) Flexibility: When considering the long run simulation, it is unrealistic to expect exclusive access to computing resources over the entire simulation time, because these resources on the grid are generally shared by many users. One must assume that each cluster will be available only for a part of the simulation time. The code should, therefore, be flexible enough to continue simulation on different target clusters dynamically.

(2) Robustness: The grid is inherently unstable and heterogeneous. To the matter worse, the more resources used for the simulation, the higher the probability of trouble events. The code should, therefore, be robust against the network/cluster trouble events, including long queuing time.

(3) Efficiency: The code should be highly parallelized to reduce the computation time.

Although several grid programming models have been proposed [17], it is difficult for these models to satisfy all the requirements at the same time. For example, grid-enabled MPI enables the code to execute efficiently, but it does not provide the mechanism for dynamic resource switching or for failure recovery.

b) Combined GridRPC and MPI strategy

As a result of the above considerations, two programming models, GridRPC [18] and MPI are used. GridRPC is designed to support RPC on the Grid based on the client-server paradigm. It has functions for dynamic execution of server programs, for detection of network/server errors, and for time-outing to avoid waiting for a long time. These functions may be used to satisfy above first two requirements.

On the other hand, MPI is used to realize efficient execution of both MD and QM simulation. In implementing the code, we used Ninf-G [19] and MPICH [20] as reference implementations of GridRPC and MPI, respectively.

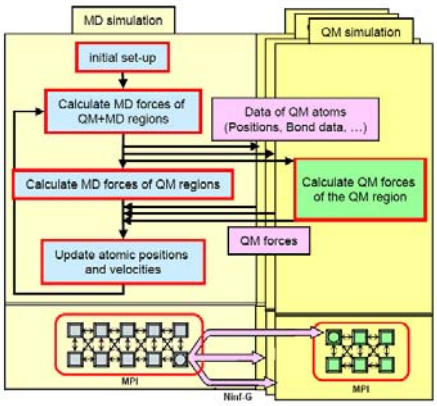


Figure 5. Execution flow of the QM/MM simulation

The execution flow of the code, divided into two parts, MD part and QM part, is depicted above (Figure 5). First of all, MD part calculates the behavior of atoms in the entire region in parallel using MPI. Then, it sends the data of QM atoms to each QM part using GridRPC. Each part calculates the force in the QM regions in parallel using MPI again. After finishing all the simulation in the QM region, MD part gathers the force data to update atomic positions and velocities. By repeating the cycle, the simulation precedes the time step. When some trouble on servers takes place, the client stops execution of the target QM simulation and try to allocate it on another cluster to restart the simulation.

c) Preliminary Results of the long run experiment

The target experiment is a simulation of the diffusion problem in a box-shaped Si system. One cluster was used to execute the MD simulation and seven clusters for QM simulation. The total number of atoms is 1728. Five QM regions were defined totally, each of which has only one QM atom. The QM simulation of each region is allocated dynamically on five clusters among seven.

The experiment continued for two weeks, in which the code tried to execute QM simulations 47593 times. Over this period, 524 trouble events occurred, such as connection failure, batch system down, and queuing time-out. Most of them (80 %) occurred during allocating QM simulation on the target cluster. But even after the allocation, there are trouble events such as exceeding CPU time limit which results in the termination of a program by the batch system.

In spite of these troubles during the experiment, the code succeeded in continuing simulation by changing the target cluster. The result clearly shows that the approach, combining GridRPC and MPI, is valid for long run simulations on the grid.

3) Time-Dependent Density Functional Theory (TDDFT) experiment

TDDFT [16] is used for molecular simulations in computational quantum chemistry. TDDFT was parallelized with Ninf-G.

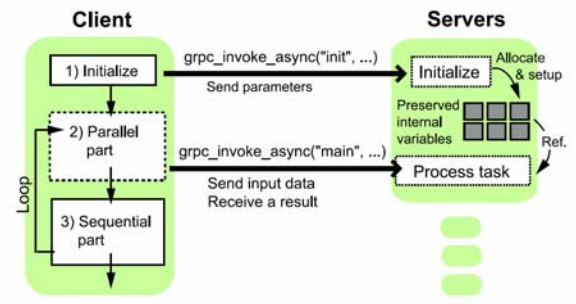


Figure 6. Implementation of TDDFT using Ninf-G

-- Implementation of TDDFT: In general, the key equation in TDDFT is converted to a linear equation by the linear approximation of the reflectance function so that it is calculated on a computer as a problem of the frequency domain. The redesign adopted a method for numerical integration of the equation because it is suitable to parallel computation. As shown in above (Figure 6), steps 1) arrays initialization, 2) time evolution of the orbital ($\varphi_i(t) \rightarrow \varphi_i(t+\delta t)$) and 3) calculation of n(r), V_H and V_{xc} with φ_i are iterated with incrementing the time step. Mostly 2) is a hotspot of TDDFT calculation which can be divided into multiple tasks to perform numerical integration for each orbital.

Implementation of TDDFT using Ninf-G calculation requires the least modification of original code. In Figure 6, φ_i and potential data are sent to the server in the asynchronous RPC for main calculation step 2) and renewed φ_i' is returned to the client. Step 3) is executed with φ_i', subsequent to completion of asynchronous calls.

-- Lessons learned: Typically, a hotspot would be more than 50% of total calculation to simulate medium size molecules. For example, the simulation of the leg and protected Au_{13} molecule, 122 RPCs are invoked at every time step and more than 5,000 iterations are required to achieve significant or practical results. The total number of RPCs will be 610,000. Because one RPC costs a few seconds on the Pentium III 1.0 GHz processor, the estimated calculation time could be one week. In addition, each RPC requires large data transmission: 4.87MB to the server and 3.25MB from the server.

The TDDFT program was parallelized within a short time and achieved reasonable performance on a single cluster system. However, some RPCs failed when the program is run in the testbed. The failure was caused by poor communication performance on the PRAGMA testbed between some sites, such as at most 300 KB/sec, less than 40 KB/sec in the worst, and varies significantly over time. Consequently, retry and timeout mechanisms of the RPC session, elimination and recovery methods of a down server were added to the client program to overcome the instability, using optional functions of Ninf-G. In the end, the modified program achieved the estimated one-week execution time. Further changes to the algorithm are necessary to reduce the communication overhead or to keep down the calculation cost for scalability on the grid.

In terms of system support and middleware requirement, the hardest part was to setup and test all client and server programs on all sites. The server programs were rarely updated after the distribution. Sometimes, manual logins to remote sites are necessary to kill zombie processes that were left by the unhandled faults and to check an error log for each server. These problems are shared by most applications and should be automated by administrators' and middleware developers' efforts.

III. END TO END GRID COMPUTING ENVIRONMENT SOLUTIONS

The end to end grid computing environments described here provide more than user interfaces such as web sites, portals or desktop clients, which improves user experience of a particular deployment strategy. They consist of components designed for specific hardware environments, instrumentation, or provide significant new enhancements to Globus toolkit, or offer new standards towards grid service interoperability.

A. Development of BioPfuga (Biosimulation Platform United on Grid Architecture)

The usual usage of the grid architecture is to run one computation on many distributed CPUs through a high-speed network. However, in order to analyze much more complicated biological systems, composed of simulations at different levels, on a new paradigm for biological science, more integrated computational approaches are required. BioPfuga is the computing grid component of the BioGrid project headed by Shinji Shimojo at the Cybermedia Center of Osaka University. The BioPfuga group developed their own biological simulation programs, which cover a wide range of fields in biological science from electronic analyses of biological macromolecules to cellome and Physiome research. In particular, AMOSS (Ab initio Molecular Orbital System for Supercomputer) for electronic state simulation [21] and pretsoX-basic for protein molecular dynamics simulations [22, 23] have been driver applications for the BioGrid architecture.

QM/MM simulation is highly suitable for the Grid. Most of biological simulation programs have been separately developed by distinct organizations. This means that it is difficult to integrate these programs into a single program. Rather, it is easy to integrate and manage these programs on a wide-area computational environment, or the grid. Also, some simulation programs require special hardware devices. This means that it is difficult to integrate these traditional programs into a single program at a single site.

BioPfuga aims to establish a research infrastructure where scientists and researchers can flexibly combine a variety of simulation programs to a multi-scale simulation program for their research purpose. To this end, grid services or web services for each simulation are developed based on the OGSI standards, and are now migrated to WSRF. In addition, a standard XML format to describe data of bio-molecular simulation, BMSML, has been developed, and abstraction of data transfer interface is achieved by utilizing the inheritance of web service interface. Wrapping each simulation with grid services has achieved integration of simulations interfaces with keeping each simulation's independence.

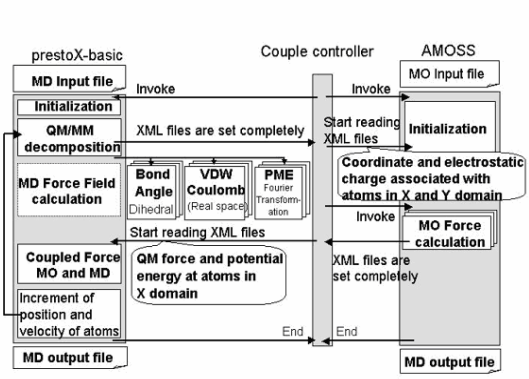


Figure 7. An example of BMSML in Base64 form for the data exchange between different computational programs for actual execution of Grid computations.

BioPfuga requires that (1) application programs be divided into a set of many pieces, each of which corresponds to a unit simulation procedure, and that (2) data communication be made between the program pieces by a standard description. For the former requirement, the simulation unit should not be too small for rapid computation so that the data communication time among different machines is minimal. For the latter problem, a simple, standard description has been developed using XML, BMSML for the data exchange between different computational programs (Figure 7).

Three types of XML description are used: a text form, a hexadecimal form, and a Base64 form. When the Base64 form is used, the size is only about 1.3 times larger than that used in the binary form. The advantage of the XML form for intermediate and output data is that any meta-data can be easily added as an attribute or as tagged information in addition to the actual computed data to be exchanged among the different application programs. It should be emphasized that the unit of data can always be provided in BMSML, so that the different application programs recognize and confirm the unit system for computation and analysis.

B. Telescience

Scientific imaging instruments are used in a variety of disciplines to gather vital data for research and study. Specifically, in the biomedical field, various types of biological imaging instruments, such as electron microscopes and light microscopes, are used everyday to acquire 2D and 3D datasets for further understanding of biological structures. Remote operation or "tele-operation" of instruments has become a popular solution for research scientists to acquire and share data across research domains separated by geographical barriers.

Many of these scientific imaging instruments such as electron microscopes, light-microscopes, and synchrotrons share similar properties and hardware functionality features. The Telescience architecture [24] uses web services and grid services to achieve interoperability, scalability, and performance within the architecture. It is multi-tiered and uses the notion of architectural fragments, which are reusable

components that describe a design pattern or a framework. A single architectural fragment describes the structure of architecture in terms of its components, also known as *roles*. Each of these components is represented as a web service, each having its own distinctive interface for interaction. These components can be composed with each other and with other reusable components to build up the framework.

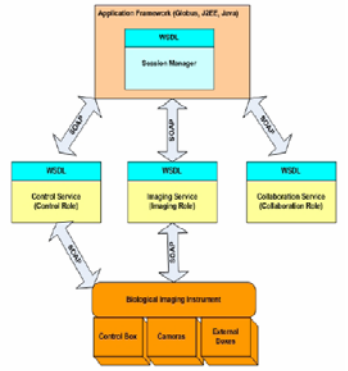


Figure 8. Multi-tier service based software architecture for telemicroscopy

Grid services are incorporated to achieve another level of integration and increased performance not possible with just web services themselves. Using grid services allows the software components to take full advantage of all the distributed resources on a Grid to effectively query, process, and store data from instruments. Finally, a set of software plug-in specification libraries is presented to describe how developers can interface with this software framework. These libraries abstract the complexities of the Grid, web services, security, and other underlying logic to the software developer in order to focus on creating a usable client interface for a particular instrument (Figure 8).

C. *MGrid and* e-*Glycoconjugates*

Molecular simulation is considered to be a promising research technique for many current and future bio/chemical research areas since the experimental methods such as X-ray and NMR spectroscopy are not efficient enough to obtain full structural information of bio-macromolecules. However, the simulations for the bio-conjugates of protein, DNA, lipid, and carbohydrates often needs much more than the computing capacity of large scale clusters or supercomputers. Simulation results on those molecules whose three-dimensional structures or appropriate simulation settings are not well-known are difficult to validate without aid of real experiments. These two critical problems, *challenging requirements of computation power and simulation results validation*, of the technique have prevented the popular and reliable use of the molecular simulations.

The MGrid system is designed to address these two issues. The system provides a shared and integrated molecular simulation grid environment for computing, databases, and analyses which consists of computational grids, data grids, and semantic grids. In addition to sufficient computing power due to grid computing, it also allows scientists to verify simulation results *in a collaborative way*, by sharing simulation jobs (e.g., input files) and results, by comparing simulation results on similar biological or chemical molecules, and by creating new simulation jobs by modifying previous ones (e.g., with different parameter values). The MGrid system is currently running on the grid testbed at the Applied Grid Computing Center of the Konkuk University (Figure 9).

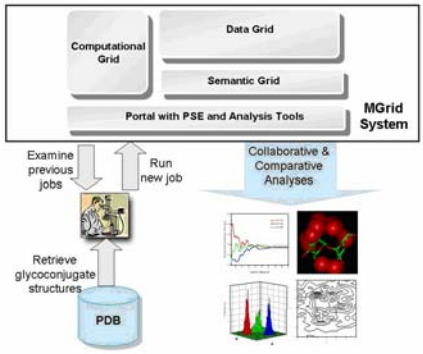


Figure 9. e-glyconjugate environment in M*Grid

e-Glycoconjugates is a grid portal for molecular simulation of glycoconjugates which is constructed by the MGrid system and designed to provide database services and analysis environments for simulation results. It is a novel alternative computational solution to overcome aforementioned two critical problems of molecular simulation for the advanced computational research on the structure and function of glycoconjugates. Services for simulation results on more than 2,000 glycan chains and 100 glycoproteins (which are so far available in structural databases such as Protein Data Bank or GlycoScience DataBase) are expected in three years.

D. *Bioinformatics grid in CNGrid and ChinaGrid*

CNGrid is supported by Chinese National Hi-tech Program (863) and ChinaGrid is supported by “211 Project” of Ministry of Education in China. CNGrid has 8 nodes distributed in CAS (Chinese Academy of Sciences) and universities. The two fastest supercomputers in China are equipped in the CNGrid. ChinaGrid integrates all of the “211 Project” universities in China is therefore much larger in scale. It uses CERNET (China Education and Research Network) to connect the computational resource in the universities in China.

The middleware of CNGrid is GOS (Grid Operating System) v2 and the middleware of ChinaGrid is CGSP (ChinaGrid Support Platform). GOS v2 is based on OGSA and CGSP is developed according to the latest grid specification WSRF. They support multiple service types, such as super (virtual) services, common web services and WSRF services. CGSP has more powerful local selection as it has many nodes consisting of CNGrid nodes and other Universities across the country.

More than 30 bioinformatics applications have been deployed to both grids, some without modifications and others with optimizations for parallel computing. For further details, please refer to [25]. Briefly, a given applications is accessible from a web portal, and an appropriate cluster or supercomputer, which may belong to different VO’s, is

selected automatically after a given request is received.

IV. Summary and Discussion

There are a number of applications deployed on the grid on a routine basis within the PRAGMA testbed. The PRAGMA workshops held twice yearly and the collaborations among PRAGMA working groups in Biosciences, Telescience, Resources and Data Computing have been instrumental in the successful deployment of these applications. While the paper by no means covers all the exciting grid activities in member institutes, the spirit of scientific collaboration and training permeating PRAGMA workshops means that knowledge is continuously produced based on our collective experiences. Some key lessons learned from these routine basis experiments are as follows:

1) The lower middleware level that deals with local cluster authentication, computation, data management has reached a level where some stability is achieved for routine basis experiments.
2) The upper middleware that deals with cross-cluster computation, virtual organizations, and meta-scheduling still require improvement. For example, it's difficult to schedule a large number of nodes simultaneously without manual intervention.
3) Some applications may be deployed without modifications though changes may be required if better performance and fault tolerance is desired.

In addition, experiences from PRAGMA members based on experiments on their own testbed also have valuable lessons. Progress is under way to deploy reusable components of the comprehensive end to end solutions within the PRAGMA testbed or at other member institutes. Some of these applications require resources beyond those available in the current PRAGMA testbed.

In order for more applications to run on the grid, there are several approaches:

1) Develop new applications with the grid in mind.
2) Execute existing applications without modifications.
3) Develop upper middleware which enable the legacy applications.
4) Realize that some applications should not be run on the grid.
5) Develop new programming models for grid access.

Many different approaches have been tried, and the grid may stay heterogeneous due to human nature. However, interoperability may be possible with availability of more open source software, and adoption of common integration technologies including but not limited to web services.

References

[1] I. Foster and C. Kesselman, "The Grid 2: Blueprint for a New Computing Infrastructure," 2 ed. San Francisco: Morgan Kaufmann Publishers, Inc., 2004.
[2] L. Smarr, "Grids in Context," in *The Grids 2: Blueprint for a new computing infrastructure*, I. Foster and C. Kesselman, Eds. San Francisco: Elsevier, 2004, pp. 3-12.
[3] Globus, "The Globus Alliance," 2004.
[4] C. Zheng, D. Abramson, P. W. Arzberger, S. Ayuub, C. Enticott, S. Garic, M. Katz, J.-H. Kwak, B. S. Lee, P. M. Papadopoulos, S. Phatanapherom, S. Sriprayoonsakul, Y. Tanaka, Y. Tanimura, O. Tatebe, and P. Uthayopas, "The PRAGMA Testbed: building a multi-application international grid," presented at CCGrid, Singapore, 2006.
[5] D. Abramson, J. Giddy, and L. Kotler, "High performance parametric modeling with Nimrod/G: Killer application for the global grid?," presented at IPDPS, 2000.
[6] O. Tatebe, N. Soda, Y. Morita, S. Matsuoka, and S. Sekiguchi, "Gfarm v2: A Grid file system that supports high-performance distributed and parallel data computing," presented at 2004 Computing in High Energy and Nuclear Physics, Interlaken, Switzerland, 2004.
[7] D. M. J. Bowman, "The Australian summer monsoon: a biogeographic perspective," *Austr. Geog. Studies*, vol. 40, pp. 261-277, 2002.
[8] D. Abramson, R. Sosic, J. Giddy, and B. Hall, "Nimrod: A tool for performing parameterised simulations using distributed workstations," presented at HPDC, 1995.
[9] N. A. Baker, D. Sept, S. Joseph, M. J. Holst, and J. A. McCammon, "Electrostatics of nanosystems: application to microtubules and the ribosome," *Proc Natl Acad Sci U S A*, vol. 98, pp. 10037-41, 2001.
[10] W. W. Li, G. B. Quinn, N. N. Alexandrov, P. E. Bourne, and I. N. Shindyalov, "A comparative proteomics resource: proteins of Arabidopsis thaliana," *Genome Biol*, vol. 4, pp. R51, 2003.
[11] H. Casanova and F. Berman, "Parameter sweeps on the Grid with APST," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. C. Fox, and A. J. G. Hey, Eds. West Sussex: Wiley Publishers, Inc., 2003.
[12] W. W. Li, R. W. Byrnes, J. Hayes, A. Birnbaum, V. M. Reyes, A. Shahab, C. Mosley, D. Pekurovsky, G. B. Quinn, I. N. Shindyalov, H. Casanova, L. Ang, F. Berman, P. W. Arzberger, M. A. Miller, and P. E. Bourne, "The Encyclopedia of Life Project: Grid Software and Deployment," *New Generation Computing*, vol. In Press, 2004.
[13] X. Wei, W. W. Li, O. Tatebe, G. Xu, L. Hu, and J. Ju, "Integrating Local Job Scheduler - LSF™ with Gfarm™," *Lecture Notes In Computer Science*, vol. 3758, pp. 197, 2005.
[14] A. Darling, "The Design, Implementation, and Evaluation of mpiBLAST,," presented at ClusterWorld, San Jose, 2003.
[15] DIANE, "DIANE framework," 2005.
[16] S. Ogata, R. K. Shimojo, A. Nakano, and P. Vashishta, "Hybrid Quantum Mechanical/Molecular Dynamics Simulations on Parallel Computers: Density Functional Theory on Real-space Multigrids," *Computer Physics Communications*, vol. 149, pp. 30-38, 2002.
[17] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz, "A Grid Programming Primer.," presented at GWD-I, GGF Advanced Programming Models Research Group, 2001.
[18] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, "Overview of GridRPC: A Remote Procedure Call API for Grid Computing.," *Lecture Notes In Computer Science*, vol. 2536, pp. 274-278, 2002.
[19] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, "Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing.," *J. of Grid Computing*, vol. 1, pp. 41-51, 2003.
[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard.," *Parallel Computing*, vol. 22, pp. 789-828, 1996.
[21] T. Sakuma, H. Kashiwagi, T. Takada, and H. Nakamura, "Ab initio MO study of the cholorphyll dimer in the photosynthetic reaction center I. A theoretical treatment of the electrostatic field created by the surrounding proteins," *Int. J. Quant. Chem.*, vol. 61, pp. 137-151, 1997.
[22] Y. Fukunishi, Y. Mikami, and H. Nakamura, "The filling potential method: a method for estimating the free energy surface for protein-ligand docking," *J. Phys. Chem B.*, vol. In Press, 2005.
[23] N. Nakajima, J. Higo, A. Kidera, and H. Nakamura, "Free energy landscapes of peptides by enhanced conformational sampling," *J. Mol. Biol.*, vol. 296, pp. 197-216, 2000.
[24] T. E. Molina, G. Yang, A. W. Lin, S. T. Peltier, and M. H. Ellisman, "A Generalized Service-Oriented Architecture for Remote Control of Scientific Imaging Instruments," presented at CCGrid, Singapore, 2005.
[25] Z. Ji, Z.-h. Lu, Y.-w. Wu, A. Guo, B. Shen, and X.-b. Chi, "Analysis of the bioinformatics grid technique application in China," presented at CCGrid 2006, Singapore, 2006.

Grid Interoperability: An Experiment in Bridging Grid Islands

Blair Bethwaite
Faculty of Information Technology, Monash University, Clayton, 3800, Victoria, Australia
blair.bethwaite@infotech.monash.edu.au

David Abramson
Faculty of Information Technology, Monash University, Clayton, 3800, Victoria, Australia
david.abramson@infotech.monash.edu.au

Ashley Buckle
Faculty of Medicine, Nursing and Health Sciences, Monash University, Clayton, 3800, Victoria, Australia

Abstract

In the past decade Grid computing has matured considerably. A number of groups have built, operated, and expanded large testbed and production Grids. These Grids have inevitably been designed to meet the needs of a limited set of initial stakeholders, resulting in varying and sometimes ad-hoc specifications. As the use of e-Science becomes more common, this inconsistency is increasingly problematic for the growing set of applications requiring more resources than a single Grid can offer, as spanning these Grid islands is far from trivial. Thus, Grid interoperability is attracting much interest as researchers try to build bridges between separate Grids. Recently we ran a case study that tested interoperation between several Grids, during which we recorded and classified the issues that arose. In this paper we provide empirical evidence supporting existing interoperability efforts, and identify current and potential barriers to Grid interoperability.

1. Introduction

Grid interoperability and Grid interoperation are two common labels often used interchangeably when discussing the challenges involved in spanning Grids. The Grid Interoperability Now community group (GIN-CG) [1] of the Open Grid Forum (OGF) [2] brings together significant experience and expertise in the area. Hence we defer to the community's definition, though others have suggested alternatives [3]. In this paper we are primarily concerned with Grids as de-centralized, cross-organizational infrastructures enabling e-Science.

A recent call for papers [4] summarizing GIN echoed the definition used within the community; describing interoperations as short term efforts addressing what needs to (and can be) done today, using as much existing technology as possible, to enable Grids to work together. In contrast, interoperability is a longer term, broader goal to achieve native interoperation of Grids and Grid middleware through open standards. In this paper we use these definitions and the umbrella term *interop* when discussion encompasses both definitions.

As e-Science continues to be embraced by the wider research community (i.e. not just the traditional drivers such as computational physics and biology) the demand for resources also increases. The capability for the simple interoperation of Grids is now extremely important for the researchers and virtual organizations (VOs) that constitute their users. Only a decade ago it was common for research institutions to have a single computational cluster, if at all, whereas now we are seeing the formulation of dedicated e-Research and e-Science centres within these institutions.

There are several benefits in achieving Grid interoperation: it increases the resources available to any one particular VO (this is especially important in countries where the national infrastructure restricts researchers' ability to 'think big'); the pooling of resources across Grids provides safety in numbers, decreasing the effect of failure within smaller parts of the combined infrastructure; peering and opportunistic usage of resources can result in better utilization, in turn increasing the cost-return ratio of the often expensive and quickly obsolete infrastructures.

Grid interoperability is a problem today for a number of reasons. Many Grids have evolved based on different and incompatible middleware (common examples include the Globus Toolkit (GT) [5], gLite [6], and UNICORE [7]). Jobs are submitted and described differently, data is stored in archives with differing interfaces, authentication and authorization are handled in customized manners, resource information is published using differing protocols and schemas, policy and usage constraints are not uniform. On top of these potential barriers, the same issues that make using Grids an often time-consuming and

error-prone process are exacerbated when attempting to cross Grid boundaries.

The case study we detail in this paper tested Grid interoperability and interoperation between Australian EnterpriseGrid [8], APAC Grid [9], PRAGMA Grid [10],[11], FermiGrid [12], and Open Science Grid (OSG) [13]. All of these Grids use some version of the GT for basic job and data management, though some have customized various components to meet specific requirements, such as Fermigrid's gateway service. These Grids were chosen partly because of existing relationships and collaborations and also because they represent a wide array of Grid maturity levels. We have deliberately avoided Grids based on other middleware stacks (there is sufficient variation between pre- and web service based versions of the GT), however this will be the subject of extended future work.

In order to ensure the validity and relevance of our study, the application we ran was 'real' science, a Nimrod/G [14] experiment in structural biology. Specifically, multiple conformations of multiple proteins from the Protein Data Bank were used in a protein crystal structure determination, using the technique of Molecular Replacement (MR). This involved testing the entire Protein Data Bank using the MR application PHASER, as discussed in more detail in the appendix. This experiment generated more than seventy thousand executions across the five Grids, requiring over half a million CPU hours.

In the following sections we discuss related work in the field, detail the methodology of our case study, classify and discuss the Grid interop issues that arose, suggest conclusions that can be drawn from this experience, and summarise the application science.

2. Related work

Previous and current interoperability work includes standardization and Inter-Grid (as in Internet). There has been considerable effort, motivated by various applications, invested in the interoperation of Grid infrastructures. This has resulted in interoperation between Grid software and been fed back into further interoperability efforts.

Standardization work is primarily motivated by the well-known need to overcome the difficulties faced by Grid users and applications developers due to the currently incompatible interfaces of the various Grid middleware and service stacks. The OGF is the de-facto leader of standardization efforts for the Grid. In early 2005 the OGF released The Open Grid Services Architecture (OGSA) (updated 2006) [15], a broad framework and foundation for current and future work (particularly standardization) encompassing the capabilities of "Execution Management, Data, Resource Management, Security, Self-Management, and Information" [15]. The OGF has working groups focusing on all these areas and a significant number of recommendations and informational documents have been released as a result of these efforts. Some of these standards, for example the Distributed Resource Management Application API (DRMAA) and Job Submission Description Language (JSDL) [16], are now supported in a variety of Grid software stacks [17]. However, in a recent summary work [17] the GIN-CG identified that despite the expertise invested and effort expended in their authoring, standards adoption by Grid middleware providers has been slow. This is compounded by the fact that in the meantime many existing infrastructures have created customised solutions to their particular problems and hence are invested in these non-standards based solutions until motivation enough for change presents itself.

The GIN initiative, as the name suggests, was started to promote interoperations on short time scales. It provides a forum for coordination of these exercises, development of best practice and guidelines, and important feedback into other OGF work including standards. APAC Grid, PRAGMA Grid, and OSG are all GIN contributors and are committed to sponsoring and enabling GIN efforts.

There has been little work on models of interoperability between Grids. Most previously suggested models are based on the concept of a unified worldwide Grid, i.e. Inter-Grid, which has been around since the definition of the Grid (as we use it today) was cemented by Foster and Kesselman in their seminal book [18]. This vision, refined later with Tuecke [19] and also in separate research [20] works, relies heavily on the resolution of existing interop problems. Of particular importance is the widespread adherence to and pervasion of standards across the Inter-Grid in order to enable higher-level services and systems. This seamless, interoperable, high-performance and high-throughput computing environment is full of promise but requires much further investigation in areas such as service brokering, negotiation and virtual market places, to name a few. It would be premature however, to embark on these activities without a thorough understanding of the existing issues.

Many production Grids have embarked on interop projects, for example OSG and TeraGrid, OSG and EGEE, EUChinaGrid [21], OSG and PRAGMA, NAREGI and EGEE, and NGS and GridPP [22]. Much of this work is ongoing. Examples of middleware efforts include GRIP [23], which resulted in an interoperability layer between UNICORE and GT2, OSG and LCG (as detailed in [24]), and the ongoing OMII Europe [25] project. These middleware and infrastructure interop projects are often pair-wise and typically involve testing

or developing interop between two specific versions of a technology, either with differing or the same vendors.

Interop work is not motivated by purely academic interest. It is the needs of applications and their users that drive progress. Most often interop work is driven by the need to increase the resource pool available for particular applications, either to exploit the aggregated size, improved availability or wider diversity. This is the case for example, with OSG and LCG interop work and federated UK-US Grid efforts (detailed further in [26]).

Much of the existing Grid interop work is characterised by the motivation to get one Grid working with another, so that users from one Grid can submit jobs to the other in a manner compatible with their current use. Related work [17],[3] has suggested that, if we continue interoperations in this fashion, each additional effort brings us another step closer to realising the goal of a unified Grid. However, whilst these efforts are important, particularly in gaining short term solutions, they are clearly not solving the whole problem, as we can see by the duplication of similar efforts around the world. It is doubtful that the continuation of this kind of interop work will lead to a uniform Grid landscape, if only because Grids are not static systems – they have a symbiotic relationship with the ever-changing science and applications that drive and utilise them, so that the requirements for the Grid systems of today will likely be highly inadequate for the Grid systems of the future.

3. Methodology

In order to distinguish interop issues from other potential problems during the case study, we followed a workflow as shown in figure 1 (below). This workflow shows the steps taken when adding a computational resource to the Nimrod/G experiment. We began the experiment on our known 'home' Grid (EnterpriseGrid) and started the resource discovery, testing, and application deployment steps on the remaining Grids in parallel. Finally, adding each of these Grids to the experiment with intervals of at least a week.

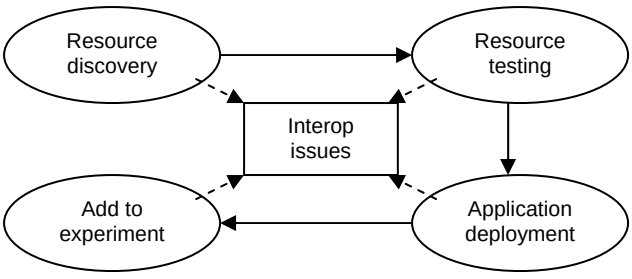


Figure 1: Interoperation workflow

Resource discovery involves finding information about resources so they can be used by Nimrod/G (further details in section 4.2) and communicating with resource providers regarding the details of the interoperation. Resource testing involves checking authorization and running automated resource checks within Nimrod/G. Application deployment (discussed further in [27]) was performed by a separate Nimrod/G experiment run once on each resource. After the successful completion of these steps, and gathering of interop issues along the way, resources from each Grid were added to the parent Nimrod/G PHASER experiment.

4. Identification of Grid interop issues

In this section we detail and discuss the interop issues encountered during our case study. We have identified five areas of interop where we encountered issues, though as we show these areas sometimes overlap and combine before exhibiting problems.

4.1. Access and security

Different Grids have varying authentication, and security mechanisms and processes, which have evolved based on the requirements of the middleware deployed on those Grids. Today the majority of Grid identity management is based on X.509 [28] public key infrastructure, of which the cornerstone is based on trust of well known third party certification authorities (CAs). This trust fabric can present problems for interoperation when crossing into a domain which trusts only a subset of the CAs from the originating domain [17]. Fortunately these sorts of problems are less common since the Tokyo Accord in 2003, subsequent formation of the International Grid Trust Federation (IGTF) in 2005, and continued growth of the member Policy Management Authorities (PMAs). During our case study we used X.509 certificates and proxy credentials signed by the APACGrid CA, which is an IGTF-AP [29] accredited CA and hence included in the IGTF CA distribution which is trusted across our testbed.

A more common issue in Grid interoperations, and even intra-operations, today is management of the virtual organisation memberships necessary for authorization within a Grid. For example, during the course of our case study it was necessary to use three separate Virtual Organization Membership Service (VOMS) [30] proxy credentials in order to be authorized to use APAC, FermiGrid, and OSG resources. Fortunately Nimrod/G binds credentials to resource definitions allowing the use of multiple certificates within a single experiment.

We believe that the use of VOMS in Grid interoperation work highlights some key weaknesses in its usage model. One of the basic assumptions in a VOMS

authorization environment is that VOs remain small (e.g. project group) and tightly bound with regard to their purpose and administration. This is a necessary characteristic to ensure minimal security risk within the VO because the existing implementation maps VOs to shared local accounts on Grid resources.

When coupled with Grid interoperations, VOMS presents a much higher security and accident risk to those within a VO established for interoperations work as they have no idea who the other members are, what their intentions are, and whether their usage could interfere with other members. Whilst VOs within production Grids such as OSG must meet high requirements for entry to the Grid (such as a minimum standard of Acceptable Usage Policy (AUP) and nominated points of contact for administration and incident response) this does not fulfil the very obvious requirement of protecting users from one another. Instead it provides a means to deal with a problem after it has occurred. Of course this does not only apply to VOs used for interoperation activities but also to any VO which is larger or more loosely coupled than the arbitrary ideal. There is an analogy here with the CA problems in the early days of Globus Toolkit development when everybody was using the Globus CA and so concern was raised over the validity of the identities asserted by it.

4.2. Resource discovery

One of the first steps when attempting to use a new Grid, after gaining authorization, is to discover and test resources. This is often a manual and very time-consuming activity. Service directories and Grid information services have gained attention recently as Grids and their user bases have expanded. Standardization of the information schemas [24] used to publish resource information and development of transformations between the existing incompatible profiles are areas of active work. For example the Grid Laboratory Uniform Environment (GLUE) working group in the OGF has produced the 2.0 version of the GLUE abstract schema.

Historically, however, information services are an area of Grids that have been neglected. For example our experiences developing and using Nimrod/G and attempting to support discovery of resource information from the Globus Toolkit Monitoring and Discovery System (MDS) led us to augment those features with user-specified knowledge and a custom user interface. This was necessary because information contained in the MDS was often missing or incorrect and when present was often not in a reliable format.

Further to the problems of incompatible information services is the difficulty of encouraging Grid managers to provide any information at all! Of the Grids in our testbed, only OSG, FermiGrid and APAC had any published information in a standard schema that could have been queried programmatically. We should note that the information required to configure a resource for use with Nimrod/G is minimal, for example for a Globus based resource the list is: the URL of the gatekeeper host, the name of the execution jobmanager, the queue name if applicable, and the operating architecture of the gatekeeper, head node, and compute hosts. Because of the lack of this information, we have built special modules into Nimrod/G that probe and query the individual resources in an attempt to discover as much of this information as possible. The fact that we were able to develop a customised solution suggests that Globus should have included a tool for populating the MDS in a consistent manner. The adaptation of monitoring tools such as SCMSWeb [31],[32], recently deployed on PRAGMA Grid, may eventually fill this gap.

It is interesting to note that the production-level Grids lead the way in uniform provisioning of information services. This is possibly because in experimental and testbed Grids the effort required to set up and support these services (which are typically not configured in an automated fashion) is seen as non-essential to the function of the resource within the Grid. This lack of resource information available in standard schemas greatly increases the human effort required to support Grid interoperations and presents a large barrier to non-expert users attempting Grid interoperations.

4.3. Usage policies and guidelines

A further preliminary step when establishing an inter-Grid testbed is to discover and ensure conformance with the usage policies and guidelines of the constituent Grids, assuming it is even possible to do this in a uniform manner. When appropriately populated information services are available much of this conformance could be gained automatically because many of the items typically covered in these documents relate to use of the execution environment (for example file system paths to temporary storage space and deployment areas).

When this information is not available it is typical to fall back to using the Grid home directory set by the job management middleware, indeed without any further guidelines this is all that is possible. This was the case for all Grids in our testbed except OSG and FermiGrid. Because of this, it was necessary to customize application deployment and write bootstrap scripts that allowed the application to be deployed and invoked via Nimrod/G in a uniform fashion.

Fortunately Nimrod/G's remote invocation mechanisms are flexible enough to allow for workarounds like this, but this is yet another area of interoperations requiring specific expert knowledge. The first step to ease this burden would be the creation, dissemination, and uptake of a middleware-independent standard job

execution environment. It is pleasing to see that the first stage in this process has begun [33] under the auspices of the GIN-CG. If this were to become widely supported it would be possible to author simple user-level tools capable of handling many of the application deployment and invocation requirements faced when running across Grids and even within Grids.

4.4. Application compatibility

As discussed in previous work [34] a common issue and barrier encountered when 'Grid-ifying' a legacy application is ensuring its portability across processor and operating system architectures. Continued improvements in both hardware and software virtualization support, and experiences with dynamically interpreted and virtual machine (VM) based programming languages, point to the medium term resolution of many of these problems. It is reasonable to expect users will see the benefit in taking an execution performance hit (virtualization overhead) several orders of magnitude lower than the throughput they gain by moving to a Grid environment. Whilst it is becoming more common to see operating system VMs used for sandboxing and consolidation benefits on Grid resources, there is still much work required before VMs are used in a user transparent manner within and across Grids to provide the sort of Dynamic Virtual Environment (DVE) proposed in [20].

This heterogeneity in operating environment concerns more than just differences in architecture and operating systems, but also issues as simple as machine configuration. Virtual memory limitations, for example, are a common problem even in intra-Grid operations let alone interoperations. During our case study we encountered several issues in this regard when moving into the inter-Grid testbed.

Our starting Grid (EnterpriseGrid) was highly homogeneous as the machines for the four sites were all purchased at the same time and from the same vendor, the compute nodes containing 8Gb of RAM and a further 6Gb of swap space for a total of 14Gb of virtual memory shared across eight processor cores. This is, relative to modern standards, a reasonably memory rich configuration for worker nodes and as such we did not encounter any memory problems. However, once we expanded execution into the PRAGMA Grid and OSG it became clear that some jobs were attempting to use more virtual memory than available on machines within those Grids.

There are two aspects to this problem. First, in many scenarios, as with our case study, it is not possible to know the resource requirements of any one particular job ahead of time and it would be unacceptably limiting to request only resources able to fulfill the upper limit when the execution profile is so varied. This means we had to accept that some jobs would fail due to exceeding virtual memory limits. We were able to augment Nimrod/G to retry these jobs on different resources, but this still resulted in the loss of many thousands of hours of CPU time (much to the disdain of those providing the cycles).

Second, as illustrated by several resources on the PRAGMA Grid during our case study, queuing and batch systems are often not fully or sufficiently configured. In many cases the configuration of limits on basic system consumables such as virtual memory and wall time were neglected, though these can be restricted in most queuing system implementations. During our case study, the memory requirements of some jobs on such resources exhausted the available virtual memory and caused the operating systems on those machines to thrash and eventually become unresponsive. Getting those machines back into operation required physically rebooting them – not a simple process when those machines are geographically distributed and under local autonomous control! This lack of configuration is not uncommon or surprising, especially given many of the resources on testbed Grids are operated and maintained by researchers who may have little system administration experience let alone expertise with Grid software. This further highlights the need for features or tools to automate and simplify the configuration of the Grid infrastructure.

4.5. Middleware compatibility

Much previous Grid interoperation work has sought to address incompatibilities in existing Grid middleware and infrastructures. This focus is mirrored in standardization efforts within the OGF; however middleware interoperability is not a silver bullet for achieving Grid interoperability. Simply ensuring that all clients and services are communicating in a standard fashion does not guarantee that those interactions will have the same effects across varying implementations.

Problems arise because these standards, such as the OGSA Basic Execution Service [35], do little to specify non-functional requirements on the interfaces, potentially resulting in unforseen side effects in the concrete implementations. We encountered an example of this during our case study when the same GT4 job submission interface had a side effect on some resources but not others.

To illustrate the problem it is necessary to understand some of the details of the Nimrod/G remote execution mechanism. Nimrod/G deploys a remote agent to computational resources which contacts the Nimrod/G server to collect work and deposit results. It is not necessary to collect the output from the Globus Resource Allocation Manager (GRAM) job which launched the agent, except when debugging the agent itself, because Nimrod/G uses its own mechanisms for file transfer. As

such, Nimrod/G does not request this output from Globus, and hence, nor can it explicitly clean it up because as far as Nimrod/G is concerned it does not exist. Also for this reason, the agent submission does not specify any particular initial working directory for the agent.

These two simple details of the Nimrod/G submission exposed an inconsistency between GRAM implementations used on different Grids. It is common for local resource managers, such as PBS and SGE, when used directly, to automatically create standard output and error files when they are not explicitly requested or specified by the job. Unfortunately the GT4 interface to PBS does not deal with these automatically generated files. This resulted in a build up of empty output files in the initial working directory of the job, which Globus had defaulted to the home directory. When coupled with VOMS (which meant the local user account was shared by many users) the eventual build up of files in this directory caused an effective denial of service for other users attempting to use that file system.

Thus, a combination of poorly specified side effects of a given interface, plus assumptions in other Grid infrastructure like the VOMS software, caused unexpected errors. This is relevant to current standardization and interoperability efforts because this issue occurred within a single software stack with an identical specification. Clearly ensuring middleware interfaces are interoperable by adhering to a single functional specification can not solve these sorts of problems. Of course, this is a general software engineering program, and is not just applicable to Grid computing. However, the wide distribution and lack of central control makes it far more significant in an open distributed system like the Grid, because the components being connected are not controlled by a single organisation.

5. Conclusions

Successful interoperation of Grids for large scale science continues to present challenges and generally requires expert technical knowledge in the field. In this paper we have discussed and illustrated some of the issues that define Grid interoperability and make Grid interoperation difficult. However, we have also shown through the application science of our case study that interoperation is possible and indeed a worthwhile pursuit. The structural biology workflow that we performed to drive interoperation efforts is an early example of techniques that are likely to become common place as emerging fields such as metagenomics continue to grow.

We identified five categories that the interop issues encountered fall into: access and security, resource discovery, usage policies and guidelines, application compatibility, and middleware compatibility. Following is a summary of the contributions of our paper.

In 4.1, Access and security, we provided evidence supporting current authentication schemes but noted that the VOMS authorization implementation deployed on some of the Grids had the potential to cause problems, especially in scenarios such as interoperation where VOs can become very loosely coupled.

In 4.2, Resource discovery, we discussed the importance of the uniform provision of information services. We also identified the need for tools which simplify the configuration and maintenance of these services, an area of potential future work.

In 4.3, Usage policies and guidelines, we noted the lack of a standard base level execution environment and showed an example use case, application deployment, which could be greatly simplified by its development and uptake. We noted that the initial development [33] of such a guideline is under way within GIN and recommend that it be pursued and published as a best practice.

In 4.4, Application compatibility, we showed that machine architecture and operating system are not the only issues requiring consideration when running an application in such a heterogeneous environment as a Grid. Good progress has been made in match-making when resource requirements are well known, but this does little to help when those requirements are highly variable and dynamic. Here we also gave examples where this was problematic for the resources as well as the application. This provides further support for future efforts to ease configuration and management of Grid computing software infrastructure.

In 4.5, Middleware compatibility, we gave an example of an interoperation problem concerning middleware that could not be solved by current standards. Many related works (e.g. [3],[17],[20],[26],[19]) have concluded with calls for the increased support of standards across various middleware. We agree with this in principle though we caution that standardization in its current form does not guarantee successful interoperations. A simple remedy to the kinds of problems we encountered might be for standards to include specification of pre- and post-conditions at interface boundaries between systems. Future work could apply formal software engineering techniques to middleware standards.

Our findings on the state of Grid interoperations are directly applicable to the areas of the GIN working groups and will, we hope, provide valuable direction for their future work. The PRAGMA Grid is actively involved in GIN efforts and we expect some of the issues we have highlighted here to spurn discussion within the PRAGMA resources working group.

6. Acknowledgements

This research was done using a large number of resources supported by various funding sources. We would like to acknowledge:

The resources provided by the Open Science Grid, which is supported by the National Science Foundation and the U.S. Department of Energy's Office of Science.;

The resources provided by the Pacific Rim Applications and Grid Middleware Assembly and its member organizations, PRAGMA member institutes are funded by funding agencies of their respective governments. PRAGMA is supported by NSF Grant No. INT-0216895, INT-0314015 and OCI-0627026.; and

The Australian Research Council's provision of the Australian EnterpriseGrid resources, including our colleagues at Deakin University, RMIT and QUT.

A great number of people have contributed in various ways and been involved in the efforts detailed herein. We would like to give particular thanks to Cindy Zheng, Neha Sharma, Mats Rynge, Nadya Williams, and Steve Androulakis.

7. Appendix: Molecular replacement Grid

The most common method of crystallographic protein structure determination is molecular replacement (MR). This technique involves using the structure of a protein that shares significant sequence similarity with the protein of unknown structure as a starting point in the structure determination (otherwise known as solving the *phase problem*). The process generally involves four steps: (1) Using sequence-based searching methods such as PSI-BLAST [36] to identify suitable structures that can be used for MR; (2) modification of structures to yield *search models*; (3) Finding the orientation and position of the search model in the unit cell of the target crystal; (4) Refinement of the model.

Molecular replacement has been used to determine the structure of approximately half of the 50,000 structures deposited in the Protein Data Bank (PDB; 67% of 2006 releases were solved by MR [37]). It is anticipated that the proportion determined by MR will grow for three reasons: First, the probability that the unknown target structure belongs to readily identifiable fold is steadily increasing, due to the rapid growth of the PDB. Second, the emergence of more sophisticated sequence searching algorithms, such as profile-profile matching [38], improve the probability of finding a suitable search model, even in cases of very low similarity (<20% identity). Third, MR algorithms consistently improve.

Where the sequence similarity between the unknown target and the search model is high (sequence identity >40%) the success rate of MR is very good, even without optimization of the search model. However, in cases where sequence similarity is low (identity <30%) MR, and subsequent structure refinement becomes non-trivial, and emphasis must be placed on the optimization of the search model. A key breakthrough in successfully applying the MR approach to situations where sequence identity is low was the development of the PHASER maximum likelihood approach [39].

Even in cases where a MR solution with low overall sequence identity can be obtained these solutions are commonly challenging to refine (the so called "model bias" trap). This situation occurs where errors in regions of the starting model cannot be adequately identified and corrected due to model bias. There are several criteria that affect the outcome of the MR calculation; 1) structural similarity between search model and target structure (measured by root mean square deviation (RMSD)); 2) percentage of residues missing from the search model (*coverage*); 3) the amount of conserved side chains that are expected to remain structurally conserved (for example in the protein interior). These factors, and thus the outcome of the MR calculation, can be influenced by improvement of the search model. The simplest approach is to remove regions of the structure that are predicted to be different in the search model and target, typically loops. However, this process is a subjective one and relies on sequence alignments, which are often incorrect, particularly at low sequence identity. Thus it is often unclear which loops should be removed and how much of the loop should be removed, and each model must be tested. In some cases the true space group is unknown and several alternatives must be tested in the MR calculation. In addition the estimated RMSD between the search model and unknown structure can affect the outcome of the MR calculation, leading in the worst case to probable solutions being missed. Therefore, the combination of multiple models, space groups and RMSD values makes MR time and labor intensive, and puts an emphasis on the availability and power of computational resources.

Utilizing PHASER discussed above, we used Nimrod/G [14] to distribute thousands of MR calculations in parallel, thereby increasing the likelihood of finding a structure. The system takes as input the structure factor data and a compressed file of search models, and parses the input and distributes jobs to available nodes on an available grid resource. Each job runs to completion independent of all other jobs, and results of the submission are returned to the user.

Using this approach we have been able to perform MR calculations using the entire PDB (~80,000 structures) in a reasonable timeframe. This feat was inconceivable three years ago (and many in our community consider it still is). Our approaches have lead to the structural solution of several key medically important proteins in our laboratories, many of which involve collaborations with researchers from Monash University and several international laboratories, which would otherwise have

remained unsolved. Much of this data is currently being published in top international journals.

8. References

[1] "Grid Interoperation Now Community Group (GIN-CG)"; https://forge.gridforum.org/sf/projects/gin.

[2] "Open Grid Forum"; http://www.ogf.org/.

[3] L. Field and M. Schulz, "Grid interoperability: the interoperations cookbook," *Journal of Physics: Conference Series*, vol. 119, 2008, p. 012001.

[4] "Journal of Grid Computing"; http://www.springer.com/computer/communications/journal/10723.

[5] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*, vol. 11, 1997, pp. 115--128.

[6] E. Laure et al., "Programming The Grid with gLite," *Computational Methods in Science and Technology*, vol. 12, 2006, pp. 33-46.

[7] D.W. Erwin, "UNICORE - A Grid Computing Environment," *Concurrency and Computation: Practice and Experience*, vol. 14, 2002, pp. 1395-1410.

[8] "Enterprise Grid"; http://www.enterprisegrid.edu.au/.

[9] "ARCS - Australian Research Collaboration Service"; http://www.arcs.org.au/.

[10] C. Zheng et al., "The PRAGMA Testbed - Building a Multi-Application International Grid," *Cluster Computing and the Grid Workshops, 2006. Sixth IEEE International Symposium on*, 2006, p. 57.

[11] "PRAGMA — Pacific Rim Applications and Grid Middleware Assembly"; http://www.pragma-grid.net/.

[12] "FermiGrid"; http://fermigrid.fnal.gov/.

[13] "Open Science Grid"; http://www.opensciencegrid.org/.

[14] D. Abramson, J. Giddy, and L. Kotler, "High performance parametric modeling with Nimrod/G: killer application for the global grid?," 2000, pp. 520-528.

[15] I. Foster, H. Kishimoto, and A. Savva, eds., "GFD-I.080 The Open Grid Services Architecture, Version 1.5," Jul. 2006; http://www.ogf.org/documents/GFD.80.pdf.

[16] "OGF Documents"; http://www.ogf.org/gf/docs/?final.

[17] Morris Riedel et al., "Interoperation of World-Wide Production e-Science Infrastructures," *Concurrency and Computation: Practice and Experience*, In-press. ; http://www.ogf.org/OGF_Special_Issue/cpeginpaper_riedel.pdf.

[18] I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, 1998.

[19] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of High Performance Computing Applications*, vol. 15, Aug. 2001, pp. 200-222.

[20] M.D.D. Assunc¸ ˜ao, R. Buyya, and S. Venugopal, "InterGrid: a case for internetworking islands of Grids," *Concurrency and Computation: Practice and Experience*, vol. 20, 2008, pp. 997-1024.

[21] "EUChinaGRID"; http://www.euchinagrid.org/.

[22] J. Jensen et al., "Practical Grid Interoperability: GridPP and the National Grid Service," Nottingham, UK: 2007, pp. 119-126.

[23] "Project GRIP"; http://www.fz-juelich.de/jsc/cooperations/grip.

[24] M. Flechl and L. Field, "Grid interoperability: joining grid information systems," *Journal of Physics: Conference Series*, vol. 119, 2008, p. 062030.

[25] "Open Middleware Infrastructure Institute (OMII) Europe"; http://omii-europe.org/.

[26] B. Boghosian et al., "NEKTAR, SPICE and Vortonics: using federated grids for large scale scientific applications," *Cluster Computing*, vol. 10, 2007, pp. 351-364.

[27] D. Abramson, "Applications Development for the Computational Grid," *Frontiers of WWW Research and Development - APWeb 2006*, 2006, pp. 1-12; http://dx.doi.org/10.1007/11610113_1.

[28] D. Cooper et al., "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," May. 2008; http://tools.ietf.org/html/rfc5280.

[29] "The International Grid Trust Federation"; http://www.igtf.net/.

[30] R. Alfieri et al., "From gridmap-file to VOMS: managing authorization in a Grid environment," *Future Generation Computer Systems*, vol. 21, Apr. 2005, pp. 549-558.

[31] "OpenSCE Project - SCMSWeb"; http://www.opensce.org/components/SCMSWeb.

[32] Napat Chalakornkosol and Putchong Uthayopas, "Monitoring the Dynamics of Grid Environment using Grid Observer," May. 2003; https://www.opensce.org/components/SCMSWeb/Members/b4305127/Document/GridObserverPoster.

[33] Morris Riedel, "Execution Environment," *OGF Gridforge GIN-CG*; http://forge.ogf.org/sf/go/doc15010?nav=1.

[34] W. Goscinski and D. Abramson, "An Infrastructure for the Deployment of e-Science Applications," *High Performance Computing (HPC) and Grids in Action*, L. Grandinetti, ed., 2008, p. 540.

[35] I. Foster et al., "GFD-R-P.108 OGSA Basic Execution Service," Aug. 2007; http://www.ogf.org/documents/GFD.108.pdf.

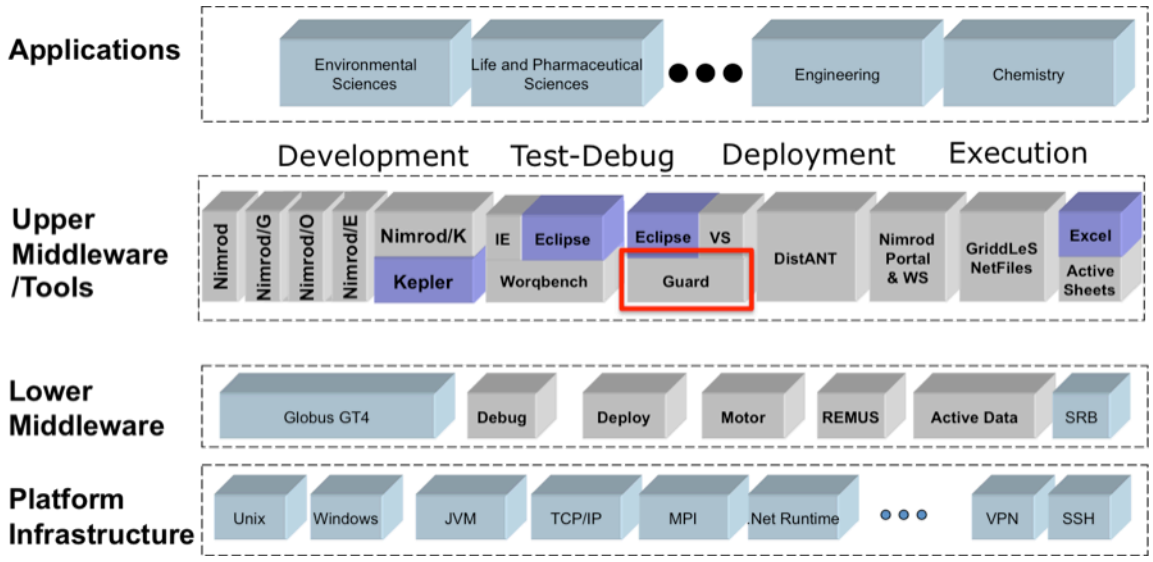
[36] S.F. Altschul et al., "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, Sep. 1997, pp. 3389-402.

[37] F. Long et al., "BALBES: a molecular-replacement pipeline," *Acta Crystallographica Section D*, vol. 64, 2008, pp. 125-132.

[38] L. Jaroszewski et al., "FFAS03: a server for profile-profile sequence alignments," *Nucleic Acids Research*, vol. 33, Jul. 2005, pp. W284-288.

[39] A.J. McCoy et al., "Phaser crystallographic software," *Journal of Applied Crystallography*, vol. 40, Aug. 2007, pp. 658–674.

Chapter 10: Guard and Relative Debugging



This chapter discusses my work in relative debugging, and its application to software engineering. (40) is the earliest major paper on this topic and discusses applications in debugging climate and pollution models. (An earlier version of (40) was awarded the best paper at the IEEE/ACM International Conference on Supercomputing in 1995.) (41) – (43) discuss the extension of relative debugging to parallel platforms. (44) shows how the same technique can be applied across multiple languages, including an experimental parallel language called ZPL. (45) discusses embedding the debugger in the Microsoft .NET framework. (46) shows how we extended relative debugging to work on dynamic data structures such as lists. (47) shows how to support relative debugging with other tools like data flow browsers and (48) discusses the architecture of the system embedded in integrated development environments. (49) is a copy of the active patent on relative debugging.

(1) Abramson D., Foster, I., Michalakes, J. and Sosic R., "Relative Debugging: A new paradigm for debugging scientific applications", the Communications of the Association for Computing Machinery (CACM), Vol 39, No 11, pp 67 - 77, Nov 1996. 10-1 .. 10-9

(2) Watson, G. and Abramson, D. “The Architecture of a Parallel Relative Debugger”, 13th International Conference on Parallel and Distributed Computing Systems - PDCS 2000, August 8 - 10, 2000. 10-10 .. 10-17

(3) Abramson, D.A., Sosic, R. and Watson, G. "Implementation Techniques for a Parallel Relative Debugger ", International Conference on Parallel Architectures and Compilation Techniques - PACT '96, October 20-23, 1996, Boston, Massachusetts, USA. 10-18 .. 10-26

(4) Watson, G., Abramson, D. “Parallel Relative Debugging for Distributed Memory Applications: A Case Study”, International Conference on Parallel and Distributed Processing Techniques and Applications June 25-28, 2001, Las Vegas, Nevada, USA. 10-27 .. 33

(5) Watson, G. and Abramson, D. “Relative Debugging For Data Parallel Programs: A ZPL Case Study”, IEEE Concurrency, Vol 8, No 4, October 2000, pp 42 – 52. 10-34 .. 10-44

(6) Abramson, D and Watson, G. “Debugging Scientific Applications in the .NET Framework”, Future Generation Computer Systems, Vol. 19 issue 5, June 2003., pp 665 - 678. 10-45 .. 10-58

(7) Abramson, D., Finkel, R., Kurniawan, D., Kowalenko, V. and Watson, G. “Parallel Relative Debugging with Dynamic Data Structures”, 16th International Conference on Parallel and Distributed Computing Systems, pp 22 – 29, August 13 - 15, 2003 Reno, Nevada, USA. 10-59 .. 10-66

(8) Searle, A, Gough, J. K. and Abramson, D. A. “DUCT: An Interactive Define-Use Chain Navigation Tool for Relative Debugging”, AADebug’03. Ghent, Belgium, September 2003. 10-67 .. 10-78

(9) Abramson, D., Chu, C., Kurniawan, D and Searle, A. “Relative Debugging in an 10-79 ..

Integrated Development Environment", Software Practice and Experience 2009, 39:1157-1183. 10-105
(10) Method for Testing and Debugging Computer Programs, Australian Provisional Patent, PM5196/94. US Patent Number 5,838,975. 10-106 .. 123

Contributions

The original work on relative debugging was performed by myself and Sosic at Griffith University, although I was the lead author for most of the joint papers with him; Foster and Michalakes provided the case studies in climate studies; Watson was a PhD student under my supervision; Finkel collaborated on the ideas in (46); Kurniawan was an B. CS. Honours student at the time; Kowalenko was a research associate under my supervision; Searle was a PhD student under the joint supervision of myself and Gough; Chu was a research associate under my supervision.

RELATIVE DEBUGGING:

A New Methodology for Debugging Scientific Applications

Accounting for discrepancies in large scientific codes, a tedious but necessary task for developers, is automated through use of the Guard relative debugger.

David Abramson,
Ian Foster,
John Michalakes,
and Rok Sosič

LARGE scientific codes are constantly evolving. Refinements in understanding physical phenomena result in changes to physics, improved numerical methods result in changes to solution techniques, and developments in computer architecture result in new algorithms. Unfortunately, this evolutionary process often introduces subtle errors that can be extremely difficult to find. As a consequence, scientific programmers can spend many hours, days, or weeks laboriously comparing the executions of two almost identical codes, seeking to identify the source of a small discrepancy.

Debuggers assist in locating program errors. They are tools that allow a user to investigate the execution state of an application program, by (for example) examining the state

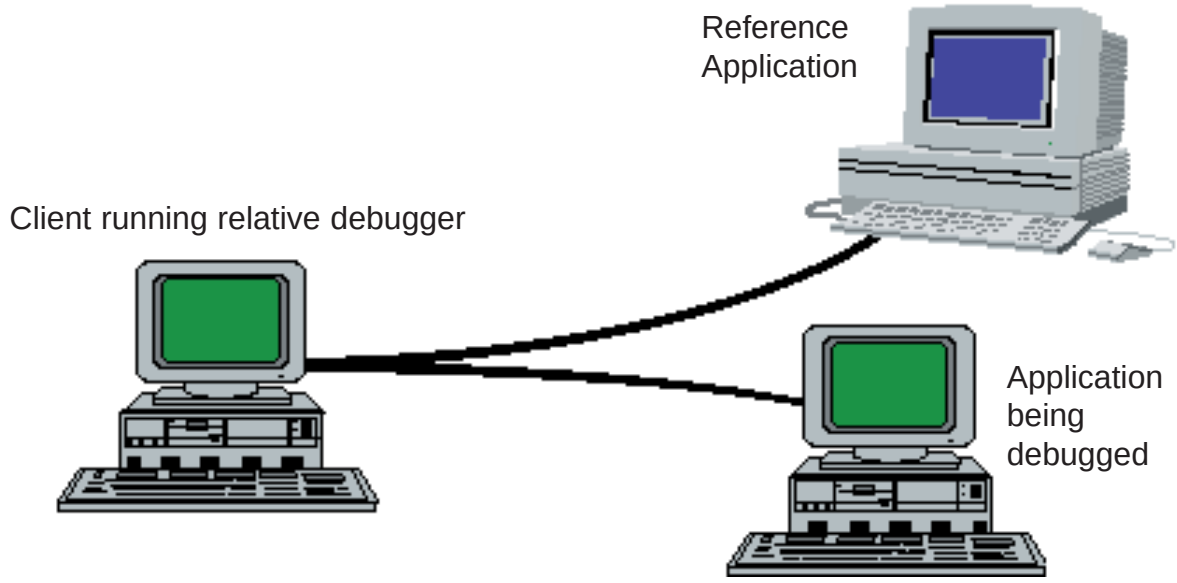


Figure 1. A relative debugger environment

of program variables. Significant recent research includes the addition of graphical user interfaces to improve the ease of use, data visualization facilities to aid the interpretation of large and complex data structures, process groups to aid the management of many independent threads in parallel machines, and support for parallel and distributed debugging [5–8].

Traditional debuggers have proved invaluable when developing new programs. However, they do not directly address the problems of maintaining and extending existing computer programs, or converting software from one machine or language to another. Programmers do not want to examine new versions of existing programs in isolation—they want to compare their execution with the execution of an older reference program that is assumed to be correct. By acting as a reference, the working version can assist in locating the section of code in the modified program that introduces incorrect values.

Existing techniques for comparing executions of two program versions are tedious, error-prone and limited in scope. For example, a programmer may invoke the two programs under separate debuggers, manually set breakpoints, run the programs, and visually compare the resulting program states. A more advanced approach is to insert output statements into both programs and then compare the output using a file comparison program [4]. This approach also has its limitations: it requires huge amounts of disk storage for the program output, involves modifications to both programs, and is not easily extended to take into account data types. For example, if floating point numbers are being compared, then programs that compare files character by character may not be sufficiently flexible.

In this article we describe relative debugging, a methodology that addresses these difficulties. The key idea in relative debugging is that errors in a new version of a program can be located by automated, runtime comparison of the internal state of the new and reference versions. When supported by appropriate tools, this approach does not require any modifications to user programs and can perform comparisons on the fly, without requiring disk storage. Comparisons can take into account differences in data representations of the two programs, making it possible for the new and reference versions to run on different machines or to be written in different languages. The latter feature is particularly important in supercomputing applications, for example, when porting a vector code to a parallel computer.

In addition to introducing relative debugging, this article describes a particular instantiation of this methodology—the relative debugger Guard—and presents the results of an experimental study in which Guard was applied to a large scientific code. Guard uses a machine-independent debugging interface to support relative debugging in heterogeneous, multilanguage environments. The experimental study involves a mesoscale atmospheric circulation model called MM5, and demonstrates the power of Guard by using an existing sequential version to account for subtle numeric differences in a parallel version.

Relative Debugging

Traditional debuggers allow a user to control a program and examine its state at any point of the execution. The user sets breakpoints in the code, interactively examines program variables, and verifies that these variables have expected values. Erroneous values can be traced to erroneous code by using information about program data flow. Relative debugging

differs from traditional debugging in two important respects. First, program variables are compared not with user expectations, but with variables in another reference program that is known to be correct. Second, because the reference program is available to compute correct values, the comparison process can be automated and the relative debugging process proceeds as follows. The user first formulates a set of assertions about key data structures in both the reference and the development versions. These assertions specify locations at which data structures should be identical: violations of the assertions indicate errors. The relative debugger is then responsible for managing the execution of the two program versions, for validating the supplied assertions by comparing the data structures, and for reporting any differences. If differences are reported, the user proceeds to isolate erroneous code by repeatedly refining the assertions and reinvoking the relative debugger. Once the erroneous region is small enough, traditional debugging techniques can be used to correct the development version. Thus, the relative debugger provides a quick and effective way of locating problems in the development version.

A relative debugger provides all the functionality of a traditional debugger, including commands for program control, state access, and breakpoints. However, the heart of the relative debugger is a set of new commands, not available in conventional debuggers. These commands support the relative debugging methodology. We introduce them briefly here, and describe them in more detail in the section about Guard.

Because the reference and development versions of the program are executed concurrently, a relative debugger must be capable of handling two programs at the same time. It is useful for the relative debugger to support the debugging of programs written in different programming languages and executed on different computers in a heterogeneous network, as shown in Figure 1. This makes it possible to use the relative debugger when porting programs from one language or computer to another. The implementation details of controlling multiple concurrent programs on a heterogenous mix of machines are outside the scope of this article, and are discussed in more detail in [3].

A relative debugger checks user-supplied assertions by comparing data structures in the reference and development versions. It performs necessary transformations of different internal data representations on different computers or in different languages. When performing comparisons, the debugger must take into account different data types, allowing for such issues as inexact equality in floating point numbers, and differences in dynamic pointer values. This aspect of the debugger will be illustrated in the next section.

If violations of assertions are reported to the user, a number of approaches are possible for reporting differences in data structures, ranging from text to advanced data visualization techniques. If there are only a few differences, the numeric values of differences are printed out. If the differences are numerous, then visualization techniques are required to

MM5

The Penn State/NCAR Mesoscale Model is a public domain limited-area primitive equation model designed to simulate meso-alpha scale (200–2000km) and meso-beta scale (20–200km) atmospheric circulation systems [1, 2]. Developed and maintained by Pennsylvania State University and the National Center for Atmospheric Research, the model is used for real-time regional weather forecasting, air-quality studies, regional climate prediction, and other areas of atmospheric research.

An MM5 domain is a horizontally-uniform rectangular grid representing three-dimensional regions of the atmosphere. The vertical coordinate system is terrain-following, with layers distributed more closely to the surface. A large number of physical parameterizations, collectively called "physics," provide forcing terms for solar radiation, cloud formation, surface and boundary layer effects, etc. Input to the model comes in the form of initial conditions—derived from observations, other simulations, or both—and lateral boundary conditions that are input periodically. The model also can also input observations that are scattered over time to further improve the quality of the simulated output.

MM5 incorporates a sophisticated nesting mechanism for mesh refinement over complex terrain or developing systems, providing the ability to resolve to a horizontal resolution as small as several hundred meters. Nests may be overlapped and even moved, if necessary, over the course of a simulation.

MM5 is written almost entirely in standard Fortran 77. (However, pointers are used to keep track of different copies of model data that correspond to different nesting levels). Loops and data structures in the MM5 code are structured for good performance on traditional vector supercomputers such as those built by Cray Research Inc. Thus, although logically the code can be thought of as invoking physics routines on a set of independent vertical columns, for maximum vectorization the code is structured so that the longitudinal loop is innermost.

References

1. Anthes, R. 1986 summary of workshop on the NCAR community climate/forecast models. *Bull. Amer. Meteor. Soc.*, 67 (1986), 194–198.
2. Grell, G.A., Dudhia, J., and Stauffer, D.R. A description of the Fifth-Generation Penn State/NCAR mesoscale model (MM5). Technical Report NCAR/TN-398+STR, National Center for Atmospheric Research, Boulder, Colorado, June 1994.

present them in a meaningful way. Examples of the different techniques will be shown in the reminder of this article.

The Guard Relative Debugger

The Guard relative debugger [1–3] provides standard debugging commands to start process execution, to set breakpoints, to examine process state, and so on. The main power and novelty of Guard, however, come from its specialized commands supporting relative debugging. These commands can be broken into two broad groups, process control and data comparison.

Program Control

Guard commands use symbolic names to refer to executing programs. Names are assigned when Guard launches a new program or when it attaches to one that is already running. Guard can handle an arbitrary number of programs concurrently, limited only by operating system restrictions.

Programs are launched using the `invoke` command, which has the following syntax:

```
invoke program_name command [machine_name]
[user_name]
```

The `program_name` is a Guard variable that is then used to identify the running program. The `command` specifies the command line for the program. The optional `machine_name` and `user_name` make it possible to run the code on a remote system as another user. In this case, Guard creates and attaches to a remote process on another computer system. The ability to execute the reference and development versions of a program on different systems eliminates errors introduced by moving the reference code to the target system.

Processes created by `invoke` remain suspended until their execution is resumed by a `verify` command, which takes a list of program names as its argument. Execution continues until the processes terminate or encounter a breakpoint or an assertion.

Data Comparison

Guard supports two different approaches to the comparison of program executions: a *procedural* approach and a *declarative* approach. We describe the procedural approach first.

The basic command for procedural relative debugging is `compare`, which has the following syntax:

```
compare <program_1>::<variable_1>
<program_2>::<variable_2>
```

This command traverses two named data structures (`variable_1` and `variable_2`) from the two named programs and compares corresponding data elements, taking into account their data type. The user can specify a tolerance value for comparisons, in which case differences are reported only if two corresponding values differ by more than the tolerance. For array comparisons, `compare` interprets array indices according to the source language. This enables comparisons of multidimensional arrays between programs whose source languages use different layouts of rows and columns, such as Fortran and C. Currently, Guard only supports the comparison of elementary variables and arrays. Support for more complicated data structures such as records and dynamic data structures is planned for the future.

To use `compare`, the user must set up breakpoints in both processes, run the programs and issue a compare command when the programs reach the breakpoints. This procedure can be tedious and error prone, especially when comparing values within loops. To eliminate this laborious procedure, Guard provides declarative relative debugging with assertions. Assertions simplify the debugging process by automating the setting of breakpoints and the comparison of data structures.

Assertions are specified by the `assert` command, which has the following syntax:

```
assert [/eps type value] [/file filename]
[/force] <program_1>::<variable_1>@
<source_file_1>:line_1 =<program_2>::
<variable_2>@<source_file_2>:line_2
```

The command contains two tuples, each with a program identifier, a variable name, a source file name and a line number. Guard plants breakpoints in the two programs at the specified line numbers, and stores the assertion information in an internal table. When breakpoints are encountered, the table is searched for matching assertions. The two data

THE KEY IDEA IN RELATIVE DEBUGGING is that errors in a new version of a program can be located by automated, runtime comparison of the internal state of the new and reference versions.

structures are then retrieved and compared. If there are no differences, then execution is resumed. If differences are detected, Guard returns to the command line, which enables the use of commands such as "print" or "compare" for examining process states.

The optional fields in the assertion make it possible to set tolerances for comparison of structures for each individual assertion (the `/eps` parameter) and to specify a file for output of the differences (the `/file` parameter). Data produced with the latter option can be used as input to a visualization program for a more detailed exploration of the differences. It is also possible to set global values for the tolerance and output files, which can be overridden using the `/eps` and `/file` parameters in a particular assertion. The `/force` option causes the contents of the data structure in the reference code to be copied into the equivalent data structure of the code being debugged if the assertion fails. This feature makes it possible to determine whether the difference which is detected by a particular assertion is responsible for the failure of the overall program without making any changes to the source of the programs.

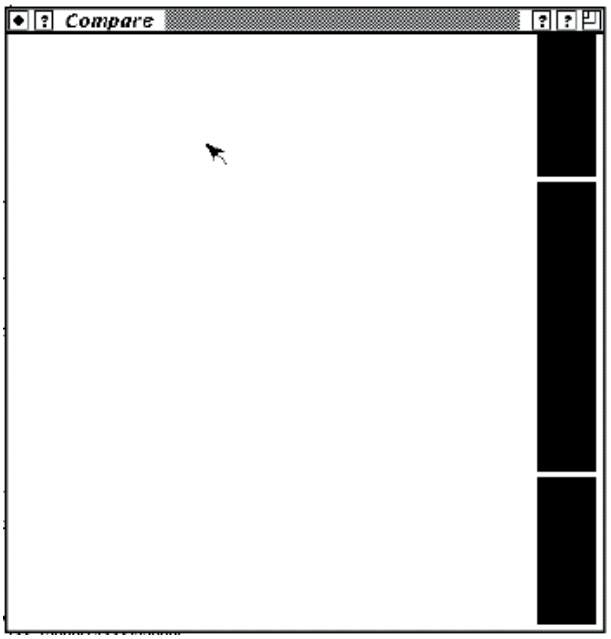


Figure 2. Visualization of boundary errors in two-dimensional arrays, produced by Guard's built-in array visualization system

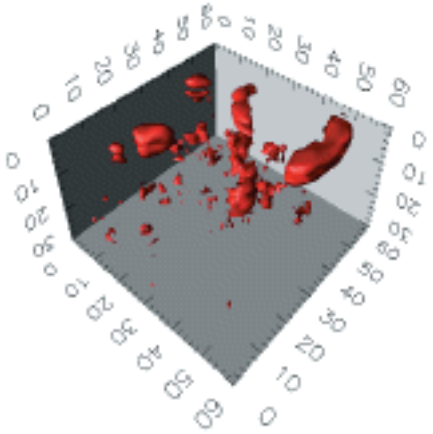


Figure 3. Error surface between data structures in two weather models, produced by Data Explorer visualization of Guard output

Subarray Expressions and Index Permutations

By default, Guard compares all elements of array data structures. However, in many cases it is desirable to consider only a subsection of each array. For example, when comparing a sequential reference code with a parallel implementation, it is often the case that additional array rows and columns are added to the parallel code to facilitate the communication of boundary values.

Guard provides subarray expressions to allow the user to make an assertion on rectangular subarrays. The following example compares one array with a subarray of the same size from a larger data structure:

```
var1[1..61][1..61][1..23]= var2[6..66][6..66][1..23]
```

Guard also supports the permutation of array indices. This feature caters for cases where the data structures being compared are equivalent, but order their indices differently. The following syntax specifies that the second array uses an index ordering that is inverted relative to the first array:

```
var1[1.61][1.61][1.23]
= var2[1..23][6..66][6..66] /permute (2,1,0)
```

This feature has proved valuable when comparing versions of a code which are optimized for a vector architecture with those optimized for RISC ones. In these cases, the inner vector loops are often moved to outer loops to improve cache performance.

Visualizing Differences

Guard supports three approaches to the reporting of differences: text, bitmaps, and advanced visualization techniques through external visualization utilities. Text output is the simplest; the actual values and differences are printed on standard output.

The second approach is more suited to array comparisons, where text output may be excessive. In this case, only two values are printed: the maximum difference between corresponding array elements, and the total cumulative difference between all elements. Most of the information is reported in a rectangular bitmap displayed on the screen. In this bitmap, white pixels denote values that are the same, and black pixels denote values that are different. This simple array visualization is particularly useful for detecting erroneous loop bounds and addressing expressions, because these types of error tend to generate regular

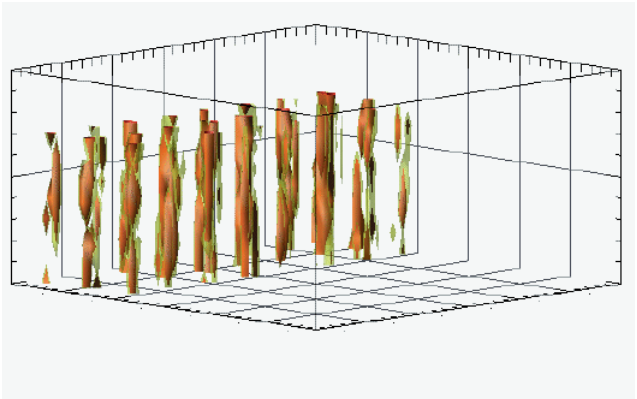


Figure 4. Error surface showing difference in ozone concentration between two photo-chemical pollution models, produced by Data Explorer visualization of Guard output

patterns on the display. An example of such a visualization in shown in Figure 2. In this case, a number of columns of the two arrays differ, and this is clearly seen as a black stripe on the right-hand side of the array visualization. Arrays with more than two indices can be folded onto two dimensions using a number of standard techniques.

```
1. invoke new mpmm
2. invoke old mm5.exe
3. eps relative float .001
4. assert old::ta[1..60] [1..60] [1..23] @SOLVE3.f:157=
   new:ta[1..23] [6..65] [6..65]@compute_solve3.f:908
   /permute (1,2,0)
```

Figure 5. Assertions used to trace errors in two models

The most powerful technique supported by Guard involves the use of commercial data exploration and visualization software such as IBM's Data Explorer (DX). A complete set of differences can be saved to a file using a parameter on the `assert` command. Values from the file can then be displayed using DX. This use of advanced visualization techniques is well suited to the display of differences in arrays with more than two dimensions. Animations can be used to convey the development of differences as the two programs execute. Figures 3 and 4 show a few examples of the visualization of errors using DX. In both of these cases an error surface is plotted to show the regions of three-dimensional space in which the error exceeds a threshold. For example, in Figure 3 the difference between the temperature variable is shown in red when it exceeds 0.1% relative error. In Figure 4, the red region indicates errors exceeding 10%, and the yellow region indicates errors exceeding 5%. This image shows that the error only occurs in half of the space, which corresponds to a region that models the physics of pollution transport over land rather than water. The picture also shows that the error is transported vertically, implicating the physics code responsible for vertical advection. Displaying the error surface in this way provides a great deal of information about the potential source of the error, including its possible location. For example, the position of the error may exclude some routines of the code that do not manipulate that region of space.

Underlying Guard Technology

Space does not permit a detailed description of Guard implementation. However, we can provide a few relevant details. Guard is implemented above a platform called Dynascope, which provides an interface for building debuggers [9, 10]. The interface contains operations for process control, state access and breakpoint handling.

Dynascope's (and hence Guard's) machine independent debugging interface is provided through debugging servers [10]. A debugging server is associated with each executing program. This server executes as a separate process or as part of the program, and is responsible for receiving and executing debugging requests from Guard. The communication between servers and Guard is performed by TCP/IP.

Because Guard is implemented in terms of Dynascope mechanisms, Guard can be ported with little modification to any computing platform supported by Dynascope. (Currently, debugging servers have been implemented for Sun, Silicon Graphics, IBM RS6000, and Next computers). In addition, the support for remote debugging allows Guard to compare programs executing on different machines. For example, we have successfully run Guard across the Internet, with Guard executing on a Silicon Graphics computer in Australia, and the two programs being compared running on a Sun computer in Europe and a Next computer in the U.S., respectively.

Test Cases: Atmospheric Modeling Applications

In this section we describe the experiments performed using Guard to trace discrepencies in MM5, a large mesoscale weather model. First, we describe an experiment in which Guard was used to determine that two sequential versions of MM5, thought to implement different dynamics, in fact provided identical results. Second, we report an experiment in which Guard was used to investigate and account for discrepancies between a correct sequential version of MM5 and a single process parallel version called MPMM. This task was particularly challenging because of the different data and code structures. More details of MM5 and MPMM are provided in the two sidebars appearing in this article.

Comparing Sequential Codes

In this experiment we used Guard to compare two versions of the sequential MM5 code. The first version was a Unix code produced at Argonne from the original NCAR Cray code; the second was a code produced at NCAR that incorporated refined versions of a number of dynamics and physics routines. The two codes were supposed to be functionally equivalent, but we did not expect bit-for-bit correspondence. Our task was to quantify any discrepancies between the two models, and to isolate the source of these discrepancies.

We proceeded as follows. Guard was configured to compare a number of the principal MM5 data arrays. Initially, we ran the models for 30 time steps with a small tolerance value. There were no errors reported. Further, the errors were all zero, indicating that the codes were producing exactly the same results. To our surprise, Guard had indicated that the two models were providing bit-for-bit identical results.

Although simple, this experiment provides some encouraging information regarding Guard's capabilities. First, it confirmed Guard's ability to deal with large, complex codes. MM5 is over 30,000 lines in length, and the data structures on which it operates are also large. Second, the ease with which we could perform the comparisons, looking for both tolerance-based differences and nonidentical values, was impressive. Third, the computational overhead of using Guard proved to be (as expected) minimal relative to the execution time of the numerical model under examination. Finally, the experiment was performed by one of the authors who had no prior knowledge of the structure or operation of MM5. In all, it required about 1 hour from the initial examination of the code to the specification of the assertions until the results were delivered.

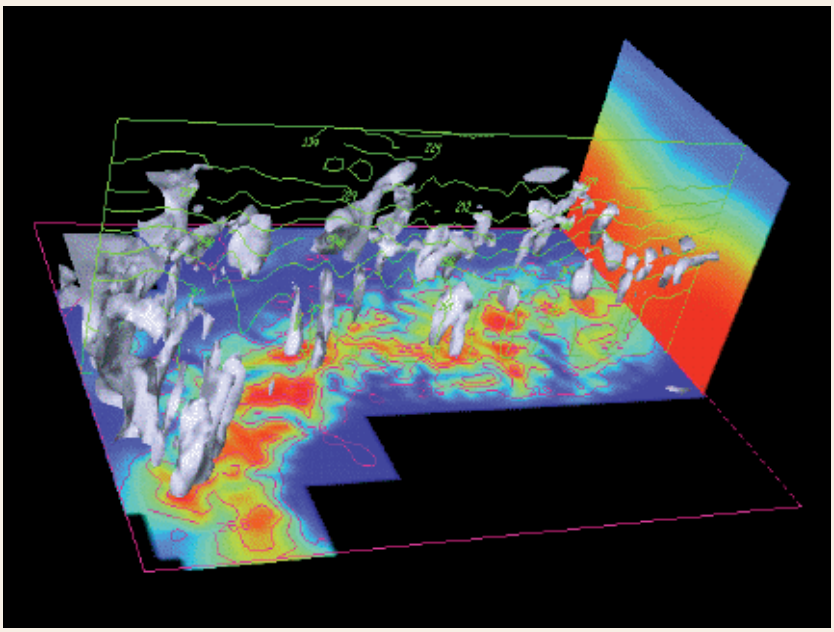
Comparing Parallel and Sequential Codes

In the second experiment reported in this article, we used Guard to compare the sequential MM5 and a

M P M M

The image in this sidebar depicts MPMM simulation of a precipitation event, January, 1991. Output is from a 5km irregularly shaped nest incorporating high-resolution terrain data for a region over the Alps. The model was run on 12 processors of an IBM SP2. The visualization is using the VIS-5D package from the University of Wisconsin in Madison.

MPMM is a version of MM5 designed to exploit scalable parallel computers [1, 2]. It is intended to be functionally equivalent to MM5 (that is, it should yield the same results), but is structured quite differently in order to execute efficiently on parallel computers. It has been developed by researchers in the Division of Mathematics and Computer Science at Argonne National Laboratories in Chicago, and although it is based on the original MM5 code, it has a large number of differences.



Two significant concerns that arise when moving a vector code such as MM5 to a scalable parallel computer are single processor performance and load balance. MPMM is structured to address both these issues. The MM5 code is restructured to obtain a column-callable form, which permits the model to be called once for each vertical model column. A portable runtime system library (RSL) is then used to invoke the column-callable model once for each column. This approach facilitates load balancing, as it is easy to move individual columns between processors in response to changing load distribution. It also results in better cache utilization. The resulting code performs well on scalable parallel computers, achieving, for example, 2 GFlops on a 64-processor IBM SP2 with Power 2 processors, on a 61-by-61-by-23 domain.

References

1. Foster, I. and Michalakes, J. MPMM: A massively parallel mesoscale model. In Geered-R Hoffmann and Tuomo Kauranne, Eds., *Parallel Supercomputing in Atmospheric Science*. World Scientific, NJ, 1993, 354–363.
2. Michalakes, J., Canfield, T., Nanjundiah, R., Hammond, S., and Grell, G. Parallel implementation, validation, and performance of MM5. In *Parallel Supercomputing in Atmospheric Science*. World Scientific, NJ, 1994.

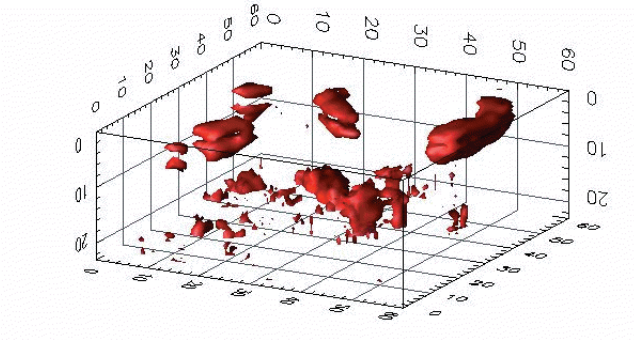


Figure 6. Temperature error surface showing error > 0.1%

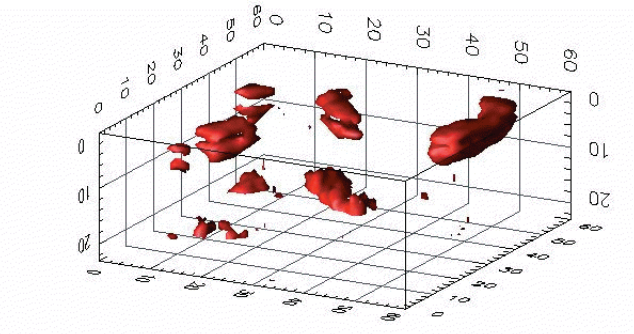


Figure 7. Temperature error surface when 13 iterations are used by both codes

single-process implementation of the parallel MPMM. As in the preceding case, we started with the assumption that the code was incorrect. The challenge was to validate this assumption and then to determine the source(s) of any discrepancies.

The preliminary set of assertions, shown in Figure 5, was applied to major data structures that showed a number of differences. Commands 1 and 2 start the two different versions of the model. Command 3 sets the tolerance to ignore minor numeric diffences and allows the user to focus on significant errors only. Command 4 specifies an assertion for comparing the data structure `ta` in the new and old models. Note that subarrays are used to extract the correct regions of the arrays for comparison, and that index permutation is required because the two arrays are ordered differently. The differences were initially small and grew slowly. Visualization showed significant spatial structure, suggesting that differences were not due solely to floating-point rounding. Incorporating further assertions allowed us to identify the planetary boundary layer scheme (subroutine HIRPBL) as the source of at least one error. Assertions applied to the input arguments to this routine showed only minor differences concentrated around the boundary of the grid; assertions applied to the output arguments showed larger and more widespread differences (Figure 6).

Having localized the error to the HIRPBL routine, we switched to a manual code inspection. We soon identified the source of the discrepancy. The problem was a difference between the ways the serial and parallel models determined the number of minor loop iterations required for a vertical column of air at each grid point. For a given point, this number of iterations can vary from a minimum of 2 to a maximum of 13 (for this problem) and the difference is subject to terrain and conditions of the atmosphere.

In order to maximize the performance on a vector supercomputer, the sequential code first determined the maximum number of minor iterations over an entire north-south strip, and then performed that number of iterations for each column in that strip. Applying this same strategy in the parallel code would introduce an unnecessary interprocessor communication (to determine the maximum number of iterations) and work. Hence, the parallel code performed only the required number of iterations for each column.

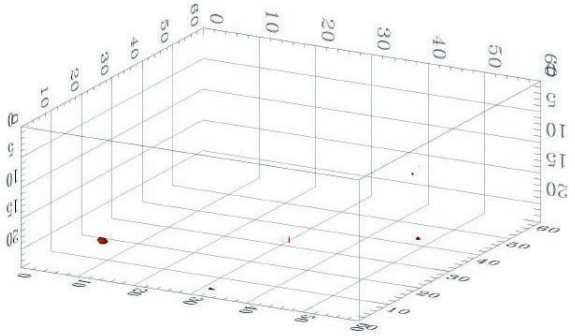


Figure 8. Temperature error surface after term added into long wave radiation term

Suspecting that this difference in behavior was the source of our problem, we modified both the sequential and parallel codes so that both performed

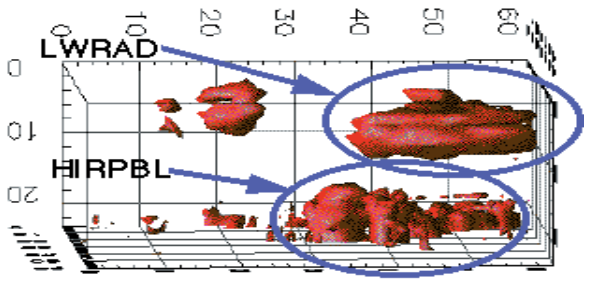


Figure 9. Front view of error surfaces identifying LWRAD and HIRPBL contributions to divergence

10-8

13 iterations on all columns. Comparison of the modified codes using Guard showed that the discrepancy over the interior of the grid had disappeared (Figure 7). This result led us to conclude that the observed discrepancy did not reflect an error in the parallel code, but rather was the consequence of a known and allowable difference in the model codes.

The modifications to HIRPBL did not remove all differences between the two models, as seen in Figure 7. Further analysis with Guard showed that there was a term missing from an equation in the long wave radiation code in MPMM (in routine LWRAD), and thus the two models were computing different amounts during this phase. When the term was added to MPMM, almost all of the remaining differences were removed, as shown in Figure 8.

Figure 9 shows a front view of the error surface after 45 time steps. Two separate regions have been highlighted. The picture clearly shows an error region at the bottom of the atmosphere, and another one at the top. As discussed, these divergences came from different regions of the model. In summary, in this experiment Guard allowed us to account for a number of discrepancies between the sequential and parallel codes.

Conclusion

Relative debugging is a new debugging methodology for applications that undergo evolutionary changes; Guard is a debugger that supports this methodology. The key idea is to provide support for the automatic comparison of program state with the state of a reference program that is known to be correct. Relative debugging appears to be particularly useful in a scientific computing context, because of the complexity of scientific models and the frequent need to adapt existing models to incorporate new physics, algorithms, or computational techniques.

We applied Guard to a large scientific code, the MM5 mesoscale atmospheric model, and obtained very satisfactory results. We were able to use Guard both to verify that two versions of the sequential MM5 were functionally equivalent, and to isolate discrepancies between sequential and parallel versions of MM5. We are currently working on a number of extensions to Guard. One important direction that will significantly increase its utility is support for the debugging of parallel programs. Another direction, motivated by the MM5 experiments described in this article, is to provide support for more flexible comparison of codes with different control structures. A commercial version of Guard is also under development.

Acknowledgments

The article reports a collaborative project between Griffith University and Argonne National Laboratory. Guard was developed at Griffith University during 1994. The experiments reported in the article were mostly performed at Argonne during March 1995. The work has allowed us to demonstrate the utility of Guard on large scientific codes, and has motivated numerous enhancements to Guard.

The development of MPMM was supported in part by the USAF and the EPA. We are grateful to Bill Kuo, Jimy Dudhia, and Georg Grell for making MM5 available to us. Some of the experimental studies were performed on the IBM SP2 at Argonne National Laboratory and the IBM SP2 at the Queensland Parallel Supercomputing Facility. We also express our gratitude to Andrew Lewis for assisting with data visualization. Development of Guard was supported in part by the Australian Research Council and IBM-Australia.

Further information about Guard is available on the World-Wide Web at http://www.cit.gu.edu.au/~davida/guard.html

References

1. Abramson, D.A. and Sosič, R. A debugging and testing tool for supporting software evolution. *Journal of Automated Software Engineering*. To appear.
2. Abramson, D.A. and Sosič, R. A debugging tool for software evolution, CASE-95. In *Proceedings of the 7th International Workshop on Computer-Aided Software Engineering* (Toronto, Ontario, Canada, July 1995), pp. 206–214.
3. Abramson, D.A., Sosič, R. and Watson, G. Implementation techniques for a parallel relative debugger. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques—PACT '96* (Boston, Mass., Oct. 20–23, 1996).
4. Galbreath, N., Gropp, W., and Levine, D. Applications-driven parallel I/O. In *Proceedings Supercomputing 93* (Portland, Oreg., 1993) pp. 462–471.
5. May, J. and Berman, F. Panorama: A portable, extensible debugger, ACM/ONR workshop on parallel and distributed debugging. *SIGPLAN Notices 28*, 12 (Dec. 1993), 96–106
6. Maybee, P. NeD: The Network extensible debugger. In *Proceedings of the Summer 1992 USENIX Technical Conference* (San Antonio, Tex., 1992).
7. Olsson, R. Crawford, R., and Ho, W. A dataflow approach to event-based debugging. *Software Practice and Experience 21*, 2 (1991), 209–229
8. Ramsey, N., and Hanson, D. A retargetable debugger. In *Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation.* (1992), ACM Press, 22–31
9. Sosič, R. A procedural interface for program directing. *Software Practice and Experience 25*, 7 (July 1995), 767–787.
10. Sosič, R. Design and implementation of dynascope, a directing platform for compiled programs. *Computing Systems 8*, 2 (Spring 1995), 107–134.

DAVID ABRAMSON (D.Abramson@cit.gu.edu.au) is an associate professor at Griffith University, Australia.
IAN FOSTER (foster@mcs.anl.gov) is a scientist at Argonne National Laboratory.
JOHN MICHALAKES (michalak@mcs.anl.gov) is a researcher at Argonne National Laboratory.
ROK SOSIC (R.Sosic@cit.gu.edu.au) is a senior lecturer at Griffith University, Australia.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© ACM 0002-0782/96/1100 $3.50

The Architecture of a Parallel Relative Debugger

Greg Watson and David Abramson

School of Computer Science and Software Engineering
Monash University, Clayton, VIC 3168
{greg,davida}@csse.monash.edu.au

Abstract

Relative debugging is a technique that addresses the problem of debugging programs developed using evolutionary software techniques. Recent developments allow relative debugging to be used on programs that have been ported from serial to parallel architectures. Such programs may change significantly in the porting process, and this raises many issues for the debugging process, such as how the array data and code transformations that occur in these situations can be modelled. Relative debugging can also be effectively used in situations where the parallel programs themselves have undergone evolutionary change. This paper presents the architecture of a parallel relative debugger, and reveals a number of novel and powerful techniques that have been developed for dealing with the issues of relative debugging in a parallel environment.

1 Introduction

Relative debugging was first proposed by Abramson and Sosic in 1994 [3]. It is a powerful paradigm that enables a programmer to locate errors in programs by observing the divergence in key data structures as the programs are executing [2][4][5][6][7]. In particular, it allows comparison of a suspect program against a reference code. This becomes a particularly valuable technique when a program is ported to, or rewritten for, another computer platform. Relative debugging is effective because the user can concentrate on *where* two related codes are producing different results, rather than being concerned with the actual values in the data structures. Various case studies reporting the results of using relative debugging have been published [5][6][7] and these have demonstrated the efficiency of the technique. The concept of relative debugging is both language and machine independent. It allows a user to compare data structures without concern for the implementation, and thus attention can be focussed on the cause of the errors rather than implementation details.

The effective implementation of a relative debugger requires an architecture that differs from traditional debuggers. The most significant requirement is the need for a client server structure, because the system must support concurrent execution and control of multiple processes on potentially different machines. Further, the debug client must be multi-threaded to allow it to receive data from a number of independent computations. Rather than using conventional multi-threading, we use an advanced data flow technique to achieve this. Because the programs might be running on different architectures, the system must also support a data format that is independent of design parameters like word size and address type. Accordingly, we use an architecture independent data format to hide machine details from the debugger.

Another important issue concerns the transformation of data structures between different versions of a program. When a program is converted from one system to another, it might be necessary to change the underlying data structures to better suite the target machine, or language. For example, one version of a program suited to a vector processor might require long single dimensioned arrays, whereas a cache based multiprocessor version might require two dimensioned arrays for efficient execution. This means that the relative debugger must be able to reverse the structure transformations seamlessly when data structures are compared. We have developed a data structure transformation algebra for the debugger that permits this seamless comparison, and we show how this is implemented. This paper presents the architecture of a relative debugger and addresses each of the issued raised above. Whilst some of the concepts have been published before [2][4][5][6][7], this paper presents a combined architecture for the first time. It also demonstrates a real implementation called GUARD-2000.

2 Relative Debugging

To the user, a relative debugger appears as a traditional debugger such as GDB [20], but also provides additional commands that allow data from different processes to be compared. The debugger is able to control more than one process at a time so that, once the processes are halted at breakpoints, data comparison can be performed. There are a number of methods of comparing data, but the most powerful of these is facilitated by a user-supplied declarative assertion. Such an assertion consists of a combination of data structure names, process identifiers and breakpoint locations.

Assertions are processed by the debugger before program execution commences, and an internal graph is built which describes when the two programs must pause, and which data structures are to be compared. In the following example the assert statement compares data from `BigVar` in `$reference` at line 4300 with `LargeVar` in `$suspect` at line 4400:

```
assert $reference::BigVar@4300 =
       $suspect::LargeVar@4400
```

A user can formulate as many assertions as necessary, and can refine them after the programs have begun execution. This makes it possible to locate an error by placing new assertions iteratively until the region is small enough to inspect manually. This process is incredibly efficient. Even if the programs are millions of lines of code, because the debugging process refines the suspect region recursively on each iteration, it does not take many iterations to reduce it to a mere screen full of code.

In more recent work, we have been examining the application of relative debugging in situations where a serial program is ported to a parallel program, or a parallel program has undergone evolutionary software development. These situations present a considerable challenge to a relative debugger. The porting process may result in significant changes to important data structures or algorithms, so the debugger must be able to take these changes into consideration when it is performing comparisons. It is also likely that the serial and parallel programs will run on different hardware platforms, so some mechanism for eliminating the effect that architecture specific information, such as byte ordering and floating point formats, may have on comparisons must be provided. The debugger must also be able to manage multiple independent processes at the same time, and these processes may reside on widely distributed systems.

The latest implementation of our relative debugger, GUARD-2000 addresses all of these issues using new and powerful techniques. The debugger supports a syntax for representing the kinds of data transformations that are likely to occur through the parallelisation of serial data structures, and for some forms of code transformations. Architectural differences are eliminated by introducing an architecture independent format (AIF) layer in the debugger, and all data is translated into this format before any comparisons are performed. The debugger also provides a simulated multi-threaded dataflow engine in order to manage data flow from the many processes that may be under the control of the debugger at any one time. Finally, the debugger utilises a client/server architecture so that processes distributed systems can be controlled.

3 Debugger Architecture

The architecture of a parallel relative debugger can be separated in to two main areas: the functions that are necessary to provide debugging services in a parallel computing environment; and those that are specifically required to support the relative debugging paradigm.

Parallel debuggers have existed ever since the introduction of parallel computing architectures. Most of these debuggers look and behave like debuggers for conventional architectures, but are usually enhanced to support the management and control of multiple processes. Many of these debuggers employ a client/server mechanism [9][12][16][21]. Some, such as DETOP [8] have been developed to support both task and data parallel codes.

A relative debugger must also provide the functionality of a conventional parallel debugger, at least internally, because in order to carry out comparisons the debugger must ensure that processes can be distributed onto multiple platforms and can be controlled independently of one another. When the relative debugging paradigm is expanded to a parallel environment, the functionality of the debugger must be extended even further. The additional functionality that must be provided by the debugger can be loosely grouped into three categories:

- *relative debugging support*, including the evaluation of user-defined assertions, and the storage and manipulation of data from executing processes;
- *parallel language support*, including the ability to interpret the parallel data structures used by data parallel language run-time systems, and *functionality* to model parallelisation techniques; and
- *parallel process support*, including the ability to control multiple independent processes, support for widely distributed processes, and support for heterogeneous architectures.

GUARD-2000 is specifically designed to operate in a parallel environment. As well as providing traditional debugging functions, the debugger also includes a range of key architectural features to address each of the functional requirements of parallel relative debugging. These include:

- a dataflow compiler and engine;
- an architecture independent data format;
- data and code transformation support; and
- a client/server architecture.

In addition, GUARD-2000 employs a technically advanced, highly modular architecture that incorporates a

multi-layered interface design, switchable network protocols and a pluggable debugger backend.

This section will discuss the architecture features of GUARD-2000 in detail, and describe how they meet the functional requirements of a parallel relative debugger.

3.1 Dataflow Engine

Central to the operation of a relative debugger is the evaluation of user-defined assertions, such as those shown in section 2. The ability to define assertions over parallel programs places a significantly increased burden on the matching and control logic of the debugger. In order to support the semantics of parallel assertions the debugger must possess the following capabilities:

- it must be possible to extract data from one process independent of another;
- breakpoints must be handled in any order, regardless of the static structure of assertions;
- it must be possible to reuse a data item in many different assertions; and
- the control mechanism must manage many processes at once.

GUARD-2000 meets all of these requirements by employing a dataflow execution mechanism [1][10] to manage the evaluations of assertions. In a dataflow environment, nodes only execute when they have tokens available on all of their operands. In GUARD-2000, the assertion statement is represented as a node that performs a comparison only when it has data available on its inputs from two processes. Data from each process is encapsulated in tokens, which are generated asynchronously as the result of a process reaching a breakpoint. Since there is no guarantee data from one process will arrive at any particular time, the available data is conveniently stored until all the arguments have been generated. Similarly if two assertions require the same data item, the token is simply duplicated and sent to the appropriate input of each assertion node.

In a normal debug session, the user will define a number of assertions about the programs under consideration. As part of its user interface, GUARD-2000 provides a compiler that converts these assertion definitions into a dataflow graph consisting of nodes that define comparison and control operations, and edges that define the flow of data through the graph. When the assertions are to be evaluated, the dataflow graph is passed to the dataflow engine for execution. To begin the execution process, the graph is "seeded" with an initial token.

Figure 3.1 shows a simplified graph that results from compiling the following assertions:

```
assert S::A[0..49]@seq.c:10 = [0]::B@par.c:10
assert S::A[50..99]@seq.c:10 = P[1]::B@par.c:10
assert S::A[0..9]@seq.c:15 = P[0]::B[0..9]@par.c:90
assert S::A[50..59]@seq.c:15 = P[1]::B[0..9]@par.c:95
```

In this example, the array `A[100]` in the sequential code has been decomposed across two slaves into the array `B[50]` in the parallel code. The first two assertions reflect this relationship. The second two assertions are comparing the first 10 elements of the sequential and parallel arrays at different line numbers.

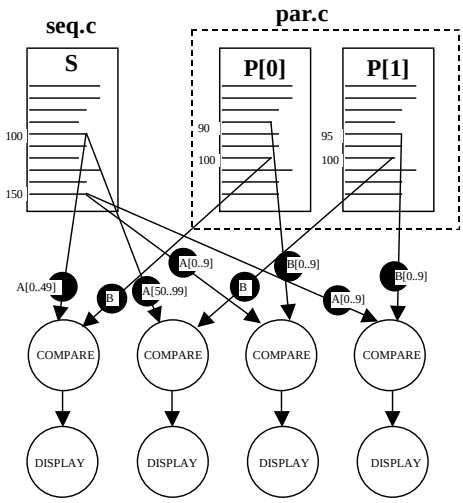


Figure 3.1: Dataflow graph generated from assertions

The resultant graph contains a compare node and a display node for each assertion statement. A compare node computes the difference between its arguments. If there is a difference, then this is reported to the user via the display node. The diagram shows tokens moving along then input arcs and arriving at the compare nodes at different times. Once a compare node has tokens on both its inputs, it will perform the comparison and then send the result to the display node.

A few debuggers have used dataflow as a control mechanism in the past [15][18]. In [18] this was concerned with using the breakpoint as an event trigger for a piece of debugger code. Likewise TotalView [21] allows the user to write expressions that are executed when a breakpoint fires, and these can be considered like dataflow expressions. GUARD-2000 extends previous work significantly by building complex tokens containing data from the processes being debugged, and then using this data in the assertion statements.

3.2 Architecture Independent Data Format

In addition to debugging programs on physically separate hosts, a relative debugger must also support programs running on heterogeneous architectures. Data from remote systems must be transferred to the client as a

result of formatting or display commands, or for arithmetic or comparison operations generated by the dataflow engine when executing assertions. The remote systems (and the client) may each employ different architectural features such as word length and byte ordering.

The problem of sending architecture specific data over a network has been addressed by standard networking protocols such as XDR [19]. Much work has also been done on the development of architecture independent file formats, with NCSA's HDF [17] now accepted as the defacto standard. However, none of these approaches address the issues of performing in-memory operations on data from architecturally different systems. As a result, GUARD-2000 provides a well-defined API and library routines that implement all the necessary arithmetic and conversion operations.

AIF is designed to achieve true architecture independence for arbitrary data types. The key components of the AIF system are:

- a format for representing data in an architecture independent manner;
- routines for converting to and from native formats;
- routines for performing arithmetic, logical and comparison operations on AIF data;
- routines for accessing components of structured data types; and
- routines for performing I/O on AIF data.

Table 3.1: AIF Format Descriptor Tags

Tag	Type
c*s*	character
i*sl*	integer
f*l*	floating-point
^*lt*	address
[*t*$_1$**]***t*$_2$	array
{*l*=*f*$_1$@*o*$_1$#*l*$_1$**:***t*$_1$**, …,** *f*$_n$@*o*$_n$#*l*$_n$**:***t*$_n$**}**	structure or union
<*e*$_1$=*v*$_1$**, …,** *e*$_n$=*v*$_n$**>**	enumeration
r*v*$_{min}$**..***v*$_{max}$*t*	range
V*l*	void
R*rt*	ZPL region
s is **s** (signed) or **u** (unsigned), *l* is size in bytes, *t* is type, *f* is a field name, *o* is field offset in bits, *n* is size in bits, *e* is enum name, *v* is enum value or range bounds, *r* is rank.	

Data is represented in AIF using two components: a data tag and a normalised bit stream. When data is converted to AIF, a series of normalisation operations are performed. These include the conversion of integers to big-endian byte ordering, characters to the ASCII collating sequence and floating-point number to the IEEE 754-1985 big-endian format [14]. The format also standardises on a byte size of 8-bits and single byte characters. Since the normalisation operations can result in the loss of native format information, the bit stream is tagged with a format descriptor string (FDS). Table 3.1 lists the current format descriptor tags.

Tags for simple data types provide AIF library routines with information such as the size of the data and whether the data is signed or unsigned. This allows different integer sizes and single, double, and extended floating point formats to be supported, and ensures that the AIF library routines can perform calculations with no loss of precision. Complex data types have tags that describe the size and memory layout of the data, and can contain nested tag types. Figure 3.2 shows an example of how a C structure is represented in AIF.

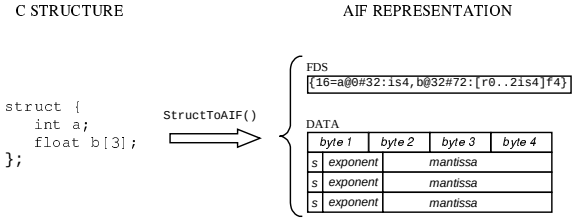


Figure 3.2: AIF Representation of a C Structure

Data is converted into AIF using one of the data conversion routines. Routines are provided to convert simple data types, such as `IntToAIF()` and `FloatToAIF()` and complex data types, such as `ArrayToAIF()` and `StructToAIF()`. All the data conversion routines return a pointer to an `AIF` structure that holds the data and tag information. This pointer can then be used as an argument to the other AIF routines. Routines are also provided to convert AIF data back to native format. Once data is converted to AIF it can then be manipulated using the arithmetic and logical operations.

Routines are also provided to access and manipulate complex data types, such as indexing elements of arrays, iterating over array indices or retrieving structure fields. In addition, routines are available to perform I/O on AIF objects, such as reading from and writing to files.

The current implementation provides descriptors for C, Fortran, and ZPL. Support for another language can be added by defining new descriptors for each of the languages data types that are not covered by the current descriptors, and by providing routines to perform appropriate operations on these data types.

3.3 Data Transformation Functions

A key component of parallel relative debugging is the ability to compare data structures between serial and parallel codes. The type of transformations that are used when a serial code is converted to execute on a parallel computer become particularly important, as the user may wish to compare data structures at various places without concern for the different organisation of the data in the two programs. In order to achieve this, the debugger must provide a mechanism to replicate the transformations that have been applied to the programs. It is desirable that this mechanism is implemented by the debugger in a way that is transparent to the user, so they can concentrate on identifying the location of the error.

GUARD-2000 supports four types of transformations to assist the user in debugging parallel codes. The transformations are specified in the debugger using a special syntax. This syntax is in turn derived from a transformation algebra that has been developed to model the different types of data transformations. The algebra and its associated syntax are beyond the scope of this paper, however of the transformation types are discussed in more detail below.

3.3.1 Data Parallel Decomposition

Except for extremely coarse-grained or parameterised models, parallelisation of an algorithm on a MIMD computer usually requires some form of decomposition of arrays in order to distribute data to the individual processes. In the case of data parallel languages, this decomposition is usually a block or cyclic decomposition, or a combination of the two, and is handled automatically by the language runtime system. In many cases, particularly where process pool sizes are dynamic, the exact partitioning is not normally known until runtime.

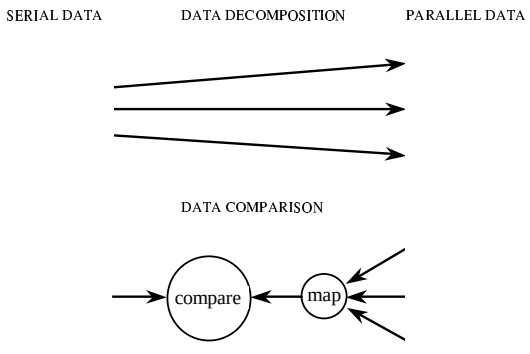


Figure 3.3: Data Distribution and Mapping

GUARD-2000 is able to deal with most of these situations using a *map* notation. This allows the decomposition model to be described in terms of two functions: a function of the array indices and a function of the processor number. When combined together, these functions form a map that can be applied to the parallel data in order that a comparison can be made. Figure 3.3 shows how this mapping process applies to data comparison.

3.3.2 Shape Transformations

It is sometimes necessary to actually change the data structures of a program in order to implement a parallel version of the algorithm. For example, data may be structured to exploit particular vector architectures and this data may need to be organised into an array to allow parallelisation. Alternatively, the shape of arrays may need to be altered to exploit processor performance improvements, for load balancing purposes, or for better cache utilisation [7].

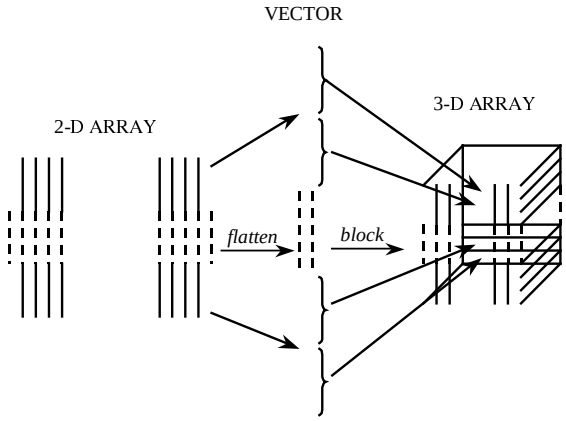


Figure 3.4: Array Shape Transformation

GUARD-2000 provides a shape transformation mechanism using standard flattener and blocker functions. The flattener function takes a multi-dimensional array and linearises it. The blocker function takes a vector and blocks it into a multi-dimensional array. By ensuring that the number of elements in the arrays always remains constant, we have a mechanism that will transform an *n*-dimensional array into an *m*-dimensional one. Figure 3.4 shows an example of transforming a 2-dimensional array into a 3-dimensional one.

3.3.3 Index Permutation

Programs that have been optimised for a vector processor may scan the data in a particular way, but this may not be appropriate for a cache based parallel machine. Accordingly, permutation of array indices is often required to improve data locality and to exploit the processor cache. In addition, permutation may also result from language differences, such as between C and Fortran. In this case, one language uses row-major order, the other column-major. The permutation necessary to compare arrays from these languages is shown in Figure 3.5.

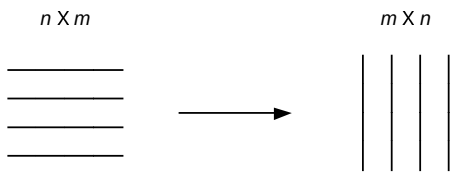


Figure 3.5: Index Permutation

Conveniently, index permutation turns out to be a special case of data distribution where the mapping is from one serial process to another. Accordingly, GUARD-2000 uses the same map notation to define index permutation.

3.3.4 Temporal Displacement

Variable re-use is a common programming construct. When this occurs, a scalar value may be calculated, used and then overwritten in a new computation, usually within some form of loop construct. However, when this type of program is parallelised, it is often necessary to promote the scalars to an array, one unique value per loop iteration. Such a variation between the sequential and parallel codes makes it extremely difficult to compare the data, because there may be no point in time in which the data exists concurrently in both programs.

Figures 3.6a and 3.6b show how a scalar variable `TEMP` is used in a Fortran code is replaced with an array `temp` in the C code. To compare these values, a copy of the scalar must be made on each iteration of the loop. When the loop has completed these copies can then be compared to the C array.

```
      DO 90, I = 1, 100
80      TEMP = X(I) * Y(I)
        X(I) = 2 * Y(I)
        Y(I) = PI * TEMP
90    CONTINUE
```

(a) Fortran Code

```
128   for (i=0; i<100; i++)
129   {
130     temp[i] = x[i] * y[i]
131   }
132
133   for (i=0; i<100; i++)
134   {
135     x[i] = 2 * y[i]
136     y[i] = PI * temp[i]
137   }
```

(b) C Code

Figure 3.6: Variable promotion example

Another common optimisation for parallel programs is to fuse a number of sequential loops into one larger parallel one. Serial codes often perform computation on entire rows, or in the case of 2 dimensional arrays entire columns, for vectorisation purposes. In a parallel program, multiple computations may be fused into a single calculation on each grid element, since in general each process will be applying the same computation to the partitioned data. Under these circumstances there will never be a point in the execution of the codes when the arrays are equivalent.

Figures 3.7a and 3.7b show sample codes where three loops have been fused into one. In this example, comparison of the array `x` prior to the execution of line 131 in the serial code and 104 in the parallel code is not possible. Instead, a temporary array must be constructed to hold the intermediate values from the parallel code, and the comparison delayed until all elements of this temporary array have been obtained, i.e. line 108 in the parallel code.

```
128   for (i=0; i<100; i++)
129     x[i] = 2 * x[i]
131   for (i=0; i<100; i++)
132     y[i] = x[i] + y[i]
134   for (i=0; i<100; i++)
135     x[i] = y[i] / x[i]
```

(a) Serial C Code

```
100   for (i=0; i<100; i++)
      {
102     x[i] = 2 * x[i]
104     y[i] = x[i] + y[i]
106     x[i] = y[i] / x[i]
      }
108
```

(b) Parallel C Code

Figure 3.7: Example of loop fusion

GUARD-2000 provides the ability to create local copies of data from executing codes, and to use this data to build multi-dimensional arrays. These local data structures can then be used as arguments to assertion statements, so that they can be compared, in the usual way, with other program variables.

3.4 Client/Server Architecture

GUARD-2000 is a multi-process parallel debugger. Most parallel debuggers support the ability to control and manipulate processes running on remote nodes, whether on a tightly coupled shared memory system or in a distributed memory cluster. Parallel relative debugging extends this paradigm further however, since the user may be debugging a combination of serial and parallel codes, all of which are under the control of the debugger at the same time.

In this architecture, each of the debug servers is responsible for managing a single process that is being debugged. The client/server architecture used by GUARD-2000 is shown in Figure 3.8. This diagram shows a scenario where the user is debugging a serial process S1 on host A and a parallel program comprising two processes P1 and P2 on hosts A and B respectively. The client/server architecture places no distinction on where the debugger client can be running, so in this case it is located on a third machine, host C.

3.5 Layered Interface Design

Both the client and server that make up the debugger are designed to be as modular as possible. To achieve this, a three-layer model has been adopted for both the client and the server. Figure 3.9 shows the components that make up the debugger.

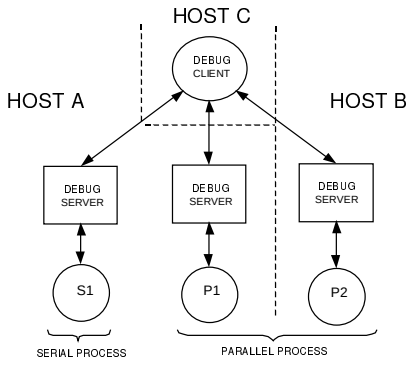


Figure 3.8: GUARD Client/Server Architecture

At the top level, the debug client provides a user interface that accepts and processes user commands in either of two modes: immediate or deferred. In immediate mode, the user interface parses and decodes commands and executes them immediately, displaying any results to the user. In deferred mode, the user interface accepts commands that are in turn passed to the dataflow compiler for translation into a dataflow graph. This graph can then be executed at a later date by the dataflow engine.

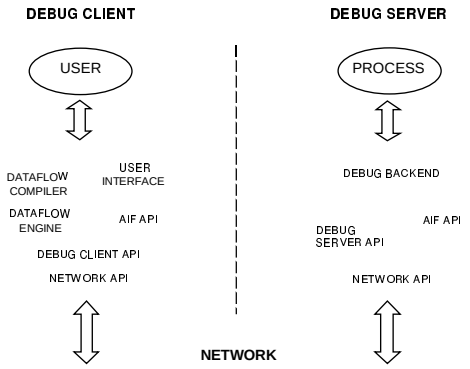


Figure 3.9: GUARD Layered Interface Model

At the next level, the debug client API layer provides a consistent interface to debugging actions that can be performed on target programs. The debug client API layer manages debug requests from the client regardless of whether they are a result of immediately executed commands, or are generated by the dataflow engine. This layer is responsible for translating these actions into the appropriate network requests using the network API layer, and for receiving and processing responses from the debug server. The AIF API provides an interface for managing and manipulating data in an architecture independent format. All data that is transmitted between client and server is first converted to this format using routines supplied by the API.

At the lowest level the network API is nresponsible for transmitting and receiving debug requests between the client and the server. A modular interface at this level allows the protocol to be tailored to suit particular requirements.

The server receives requests from the client via the network API layer. These requests are then passed to the debug server API layer, which converts the request into a form suitable for the debug backend. The backend controls the low-level architecture specific functions of the debugger. Maintaining this distinction between the server API and the backend allows for a clean separation between the architecture independent and architecture dependent parts of the server, which in turn ensures that additional architecture support can be easily added to the debugger.

3.6 Pluggable Debugger Backend

A major consideration in the design of modern debuggers is one of portability. Most recent debuggers are designed to be non-vendor specific, and support as wide a range of computer architectures as possible. The development of GUARD-2000 has been primarily concerned with the technique of relative debugging. In order to avoid the additional overhead of developing a low-level debug engine that supports multiple architectures, GUARD-2000 instead provides a well-defined interface between the debug server and the low-level debug functions. This then allows a pre-existing multi-purpose debugger, such as GDB [20], to be used to provide the low-level functionality.

The use of a backend debugger also has other benefits. It allows language and language run-time specific details to be separated from the high-level debugger and isolated in the backend. Accordingly, the debug client and server can be designed to be language independent, relying on the backend to interpret the syntax and semantics of individual languages. Support for a new language can then be reduced to a process of modifying the backend to interpret the specific syntactic and semantic details of the language.

Most data parallel languages rely on the language run-time system to manage the distribution of parallel arrays in a manner that is normally hidden from the user. The user need not be concerned about how blocks of data will be decomposed and mapped to processes (although some languages, such as HPF [13] and FORTRAN D [11] do provide mechanisms to specify this). Similarly, it is a design philosophy of GUARD-2000 that the user should interact with the debugger in terms of the data structures themselves, without concern for their decomposition and distribution. In order to enable this functionality the debugger must inherently understand how the parallel decomposition and distribution takes place. By isolating the parallel language support in the debugger backend, the

addition of new languages is simplified, while the overall debugger architecture remains clean.

4 Conclusion

In this paper we have exposed the architecture of a relative debugger, and have illustrated it using the design of GUARD-2000. The architecture has been structured to provide conventional debug primitives on both sequential and parallel platforms, as well as to provide support for relative debugging. The new features of the debugger include a dataflow matching engine, architecture independent data representation and a powerful algebra for representing how data structures are changed when programs are ported from a sequential machine to a parallel one.

We are currently exploring a number of enhancements to GUARD-2000. First, it currently uses a command line interface that cannot take advantage of sophisticated Interactive Development Environments (IDEs). Combining these would aid programmer productivity by providing an integrated interactive relative debugger along with the other IDE tools that are normally available. Second, a user currently formulates assertions by examining the programs under consideration and by following the data-flow of the variables manually. This information is then used to create the assertions. However, it would be much simpler and less error-prone to build assertions if an interactive browser could present the data-flow of a program graphically, and in particular show the define and use points for variables. Such a tool set would need to be integrated with the IDE and the debugger. For both of these we are exploring the use of the Microsoft Visual Studio environment.

Acknowledgements

This work has been supported by the Australian Research Council under the Large Grant Scheme, and by Microsoft Corporation.

References

[1] Arvind, L. Bic, and T. Ungerer, "Evolution of Data-flow Computers", Chapter 1, "Advanced Topics in Dataflow Computing", Prentice Hall, 1991.

[2] Abramson, D., Sosic, R., "A Debugging Tool for Software Evolution", CASE-95, *7th International Workshop on Computer-Aided Software Engineering*, Toronto, Canada, July 1995, pp 206 – 214.

[3] Abramson, D., Sosic, R., "Relative Debugging Using Multiple Program Versions", Key Note Address, *8th International Symposium on Languages for Intensional Programming*, Sydney, May 1995.

[4] Abramson, D., Sosic, R., Watson, G., "Implementation Techniques for a Parallel Relative Debugger", *Proceedings of PACT '96*, Boston, October 1996.

[5] Abramson, D., Sosic, R., "A Debugging and Testing Tool for Supporting Software Evolution", Automated Software Enginerring 3, 369 – 390 (1996).

[6] Abramson D., Foster, I., Michalakes, J., Sosic R., "Relative Debugging and its Application to the Development of Large Numerical Models", *Proceedings of IEEE Supercomputing 1995*, San Diego, December 95.

[7] Abramson, D., Foster, I., Michalakes, J., Sosic, R., "Relative Debugging: A New Methodology for Debugging Scientific Applications", Communications of the ACM, Vol. 39, No. 11, 69 – 77 (November 1996).

[8] T. Bemmerl, R. Wismüller, "On-line Distributed Debugging on Scaleable Multicomputer Architectures", *High Performance Computing and Networking, Volume II: Networking Tools*, Volume 797 of *Lecture Notes in Computer Science*, Springer-Verlag, April 1994, pp 394-400.

[9] D. Cheng, R. Hood, "A Portable Debugger for Parallel and Distributed Programs", *Proceedings of Supercomputing '94*, November 1994, pp 723 – 732.

[10] J. Dennis, "The Evolution of "Static" Data-flow Architectures", Chapter 2, "Advanced Topics in Dataflow Computing", Prentice Hall, 1991.

[11] G. Fox, et al., *Fortran D Language Specification*, Center for Research on Parallel Computation, Rice University, CRPC-TR90079, December 1990.

[12] P. Kacsuck, J. Cunha, G. Dózsa, J. Lourenço, "A Graphical Development and Debugging Environment for Parallel Programs", *Parallel Computing*, 22(13), 1997, pp 1747-1770.

[13] *High Performance Fortran Language Specification*, High Performance Fortran Forum, Version 2.0, Center for Research on Parallel Computation, Rice University, CRPC-TR92225, January 1997.

[14] IEEE Std.754-1985, Binary Floating-Point Arithmetic, IEEE, Piscataway, N.J., 1985.

[15] J. Lumpp, T. Casavant, H. Siegel, D. Marinescu, "Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems", Proceedings of the 10th International Conference on Distributed Computing Systems, IEEE, 1990, pp 476 – 483.

[16] J. May, F. Berman, "Panorama: A Portable Extensible Parallel Debugger", *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, May 1993, pp 96-106.

[17] NCSA HDF Specification and Developer's Guide, The National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, November 1993.

[18] R. Olsson, R Crawford, W. Ho, "A dataflow approach to event-based debugging", Software-Practice and Experience, 21(2), February 1991, pp 209 – 229.

[19] R. Srinivasan, XDR: External Data *Representation Standard*, RFC 1832, Sun Microsystems, Inc., August 1995.

[20] R. Stallman, *Debugging with GDB – The GNU Source Level Debugger*, Edition 4.12, Free Software Foundation, January 1994.

[21] *TotalView™ multiprocess debugger Users' Guide Version 3.7.7*, Dolphin Interconnect Solutions, Inc., Revision 8, October 1997.

International Conference on Parallel Architectures and Compilation Techniques - PACT '96, October 20-23, 1996, Boston, Massachusetts, USA

Implementation Techniques for a Parallel Relative Debugger

D. Abramson
R. Sosic
G. Watson §

School of Computing and Information Technology
Griffith University
Kessels Rd, Brisbane,
Queensland, 4111
{davida, sosic}@cit.gu.edu.au
Phone: +61-7-875 5049
Fax: +61-7-875 5051

§ Division of Information Services,
Griffith University.
g.watson@ins.gu.edu.au

Abstract

This paper discusses a new debugging strategy for parallel programs, called parallel relative debugging. Relative debugging allows a user to compare the execution of one program to another, and this can be used to trace errors. This technique has been found to significantly aid in problem determination. A prototype sequential relative debugger, called Guard, has already been constructed and has been used in a number of real world situations. However, the control logic it uses is not sufficiently powerful to support the debugging of parallel applications. In this paper we describe how dataflow can be used to provide a very rich control mechanism that is well suited to the parallel environment. We illustrate the system by a worked example.

1. INTRODUCTION

The development of massively parallel computer architectures and languages has created a serious problem for the software industry. Many applications demand the high performance which can be achieved through parallel systems, however, porting code to these machines can be extremely difficult. The high cost of software development means that it is desirable to make use of as much of the existing code as possible. However, this poses a number of problems. For example, a program may need to be translated to another language, and may execute on a computer architecture which is radically different from the one on which it was originally developed. Many parallel systems require specific algorithmic changes in order to exploit their performance advantages, and these may require substantial changes to the original source code. The variation in machine architectures means that the code may need to be implemented using a shared memory, message passing or data parallel paradigms, or a combination of the three. At present, it is difficult to maintain one source program across all these paradigms, thus there is the need to build, debug and maintain many different versions.

Debugging parallel programs is also a difficult task. There has been some progress in developing debugging tools which understand the management of many co-operating processes [6, 8, 9, 11, 12, 13, 15, 16, 18, 19, 20, 21, 24]. All of these tools are based on the premise that the programmer understands the detailed operation of the code and that it is possible to detect errors by comparing the behaviour of the program with the behaviour expected by the programmer. Whilst this approach is effective, it has a number of serious drawbacks. First, the programmer who is porting the code may not be the author of the original version. Thus, they may not have a good understanding of all of the algorithms and data structures. Second, it relies on the programmer to formulate an expected model of execution, which is difficult for large complex programs.

The traditional approach to debugging a program which produces different (possibly incorrect) output on two platforms is somewhat laborious. A user may invoke a conventional debugger and examine the new code in isolation, effectively repeating the debugging phase which occurred when the program was written. This approach has many inherent problems. For example, it may be some time since the code was written, and thus the programmer may not recall all of the details of the program. Further, the person performing the port may not be the original programmer, and thus may not be familiar with all of the details of the code. This situation also arises when the code has been built by a team. Another

approach is to instrument both the new code, and the original one with output diagnostics, and then to perform either a manual or automatic comparison of data values. This approach is extremely time consuming and error prone when the codes manipulate large amounts of data. If the error does not occur quickly, it may be necessary to invoke many debugger commands manually.

In 1993 we introduced the concept of a *relative debugger*,[1] a tool which assists a user in debugging a program by using another one as a reference version [1, 3, 4, 5]. Relative debugging differs from conventional debugging because the user need not be concerned with the exact state of a process in absolute terms, but simply relative to another program. For example, rather than examining each item of a large floating point array, and deciding whether the values are *correct*, the user *compares* the array in one program with its equivalent in the reference code. The result of the comparison allows the user to determine whether the array is correct without knowing the exact data values. There are two ways of comparing data, and these are discussed in detail in Section 2. First, it is possible to use *declarative* (a priori) specifications that certain data items must be equivalent. Second, an *imperative* comparison can be invoked by a user on demand. These two modes provide considerable flexibility and dramatically simplify the process of debugging.

This paper discusses the issues in implementing the control logic of a relative debugger. We show how a relatively simple method is adequate for most sequential programs, and discuss its implementation in a prototype called Guard. However, this simple mechanism does not handle the complexities of arbitrary parallel codes. We then show how a dataflow execution mechanism can be used to control parallel programs.

2. GUARD - A RELATIVE DEBUGGER

Guard is a sequential relative debugger developed at Griffith University in Brisbane Australia. Guard uses a client server architecture to allow the user to execute the debugger and the two applications on three different platforms, as shown in Figure 1. Guard behaves as a client and interacts with the user. Each of the applications is controlled by a debug server, which then communicates with the client by a remote procedure call. This separation of function means that Guard can be written in a machine independent way and machine specific information is contained in the servers.

When debugging a program, the user first determines whether the contents of various data structures are incorrect, by comparing the data values at key points in the reference code and the one being debugged. The user then interactively refines the region being examined until the cause is located. We have used this techniques in a number of real world case studies, including a Global Climate Model (GCM) [1] and a scientific pollution modelling code [17].

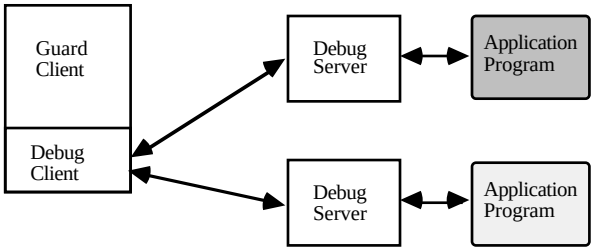


Figure 1 - Client Server Architecture of Guard

Guard provides two ways of comparing data. *Assertions* are a priori statements provided by the programmer about which data structures should be equivalent and at which points in the execution of the code. Because it is difficult to provide good temporal specifications which define when data structures should be equivalent, Guard uses line numbers. Assertions are implemented by planting breakpoints at all of the specified lines in each of the applications. When the breakpoints are triggered, Guard extracts the data from each process and performs comparison automatically. The form of an assertion is as follows:

assert process1::variable1:file1@line1 = process2::variable2:file2@line2

This assertion states that variable1 in process 1 (which will have been started on a particular machine) at line1 in file1, should be the same as variable2 in process2 at line2 in file2. It is the responsibility of the debugger to check the assertion each time the two lines are encountered. The mechanism for performing this will be discussed later in the paper.

Imperative comparisons are statements which can be issued by the programmer when the two programs are suspended. Typically, the user plants a number of breakpoints in each program and then executes them until they are triggered. It is then possible to compare any data structure between the two programs, or even within the same program. A useful combination of assertions and imperative comparisons consists of waiting for an assertion to fail, and then manually issuing a number of comparison statements to determine the cause of the violation. Imperative comparisons have the following form:

compare process1::variable1 process2::variable2

Differences in data structures can be reported through a number of techniques. First, values of scalars and small arrays can be written to standard output in text format. Second, Guard provides a basic graphics facility which can display differences in two dimensional arrays using a pixel map. Black pixels are used to denote which elements are different and white pixels are used to show which elements are the same. This mode is useful for detecting patterns in the data. For example, if a loop

[1] Patent Pending

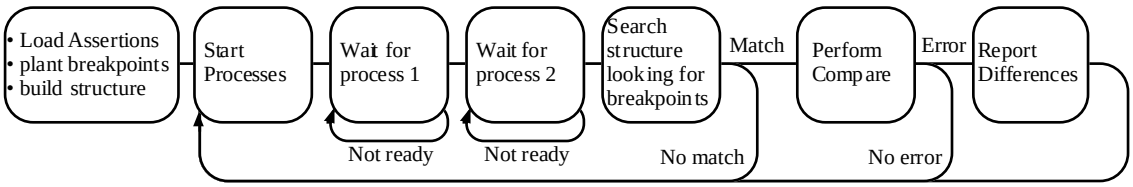


Figure 2 - Control Flow for sequential debugger

terminates early, then a line of pixels is visible in the error report. Finally, the difference in the data can be exported to an external scientific visualisation package. We have experimented with a number of displays using IBM's DX product.

Tolerances are used for inexact equality of floating point number. Thus, it is possible to set a tolerance below which errors are not reported. Given the variation in floating point behaviour on different platforms this mode can be used to detect errors only when the difference becomes significant.

In order to compare data structures, they should either be type conformant or amenable to type coercion. Further, arrays should be of the same shape and size. Guard provides a sub-array syntax for comparing sub sections of larger arrays; this is particularly useful when comparing parallel numeric programs with their sequential counterparts because the parallel arrays may contain extra rows and columns for supporting the parallel algorithm. Guard also provides an index permutation function, which means it is possible to compare arrays in which the indexes are ordered differently. Guard allows the two programs to be written in different languages, and understands the way in which various languages layout their data. We are currently producing a version of Guard which understands the data layout rules of an experimental data parallel language called ZPL [14].

3. CONTROL MECHANISM FOR SEQUENTIAL PROGRAMS

Imperative comparisons are similar in nature to the commands found in conventional debuggers like gdb [25]. They are executed immediately after being issued and use the *current* state of the processes at that time. Thus, the compare statement is relatively easy to implement because it can use existing debugger infrastructure. It must be able to extract the data from two processes and perform a difference operator on the data structures. The ability to perform this comparison in a heterogeneous distributed environment is supported by the client-server nature of Guard.

However, the assertions described in the previous section require more support by the debugger because they are specified declaratively before the codes are executed. Consequently, they do not act immediately on the executing processes. To implement assertions the debugger must:

- Save information for each assertion, which includes two sets of data structure names and breakpoint information
- Set breakpoints in both programs at the appropriate locations
- Execute both programs concurrently, possibly on different platforms
- Determine when to *fire* assertions
- Extract the data from the two programs
- Perform a comparison

In other words, assertions can be thought of as triggers for imperative compare statements. In this paper we address the issue of the control logic which is necessary to fire assertions. Assertions must be saved in a structure prior to program execution, and must be evaluated when breakpoints are encountered. When both the reference program and the code being debugged are sequential, a relatively simple control mechanism can be used for detecting which assertions are ready, as shown in Figure 2.

In this scheme, the debugger executes a sequential match process. Both processes are started on their respective machines, and then the debugger waits for a breakpoint from the first process. When the breakpoint is encountered, the debugger saves the current breakpoint address and waits for the second process. Only once it has a pair of breakpoint addresses does it search the assertion list. If it finds a match, then it performs the associated comparison. If the pair of addresses does not find a match in the assertion structure then both processes are resumed without a comparison. In this case, the processes clearly executed a path which was not expected, and so it is not sensible to perform any comparison. In this case the debugger reports the anomoly and continues execution. Likewise, if either program fails to meet a breakpoint at all then the debugger will not perform any comparisons.

The scheme described above has been implemented in Guard and has been used for a number of trials. It is effective for handling a number of unrelated assertions which have different breakpoint addresses. It can also support assertions which share breakpoint addresses, providing they all have the same addresses in both processes. However, if two assertions share only one breakpoint, then the logic fails to match correctly, because the data from one process is required for two different comparisons. Further, it requires a significant

modification if the debugger is used for parallel programs, which will be discussed in the next section.

4. DEBUGGING PARALLEL PROGRAMS

4.1 Nature of parallel programs

As discussed in section 3, the current sequential matching logic is adequate when both programs are sequential, providing there are certain restrictions on the way assertions share breakpoint addresses. However, in practice this can be restrictive when it is necessary to compare a data structure from one point of the reference code with two different places in the code being debugged.

More importantly, parallel programs do not adhere to strict sequential control patterns. Consider the comparison of a sequential reference code (*seq*) with a parallel implementation which uses two slaves (*p1* & *p2*), as shown in Figure 3. The sequential code maintains an array A[100], which is decomposed across the two slaves into two arrays, B[50]. Accordingly, sections of the array A in the reference code must be compared with each of the B arrays from the slaves at line 100, using the following two assertions:

assert *seq::A[0..49]:seq.c@100 = p1::B:par.c@100*

assert *seq::A[50..99]:seq.c@100 = p2::B:par.c@100*

However, if these two assertions are treated independently, then they will fire at different times. Even worse, the first assertion will capture A when *seq* reaches line 100, and then will restart both *seq* and *p1* Thus, the second assertion will only capture A when *seq* reaches line 100 for a second time. In order to realise the correct semantics, it is necessary to indicate that the two should behave as a single assertion, and the data structure A should be captured in the process *seq* at the same time. An enhancement of the following description language achieves this requirement, by specifying that the comparison should be performed with the same instance of A:

assert *(seq::A[0..49]:seq.c@100 = p1::B:par.c@100) && (seq::A[50..99]:seq.c@100 = p2::B:par.c@100)*

However, it is clear that the simple control logic discussed in section 3 is unable to support the required semantics, because the array A must be captured in one atomic operation, and then used in two subsequent comparisons. But the two processes p1 and p2 may arrive at their breakpoints out of synchronisation, and thus A must be saved until it has been used in both comparisons. In section 4.2 we describe a mechanism which can support the required semantics.

4.2. A Parallel Control Mechanism

The discussion to date has highlighted that in order to support parallel relative debugging, the control mechanism must have the following attributes:

- It should be possible to extract data from one process independent of another.
- Breakpoints must be handled any order regardless of the static structure of the assertions.
- It should be possible to reuse a data item in many different assertions
- The control mechanism needs to be multi-threaded to manage many processes at once.

It is possible to provide all of these attributes by using a dataflow execution mechanism [7, 10]. In a dataflow environment, nodes (or functions) only execute when they have all of their operands. The *assertion* statement can be considered as a function which compares data only when it has data from two processes. Moreover, the data can arrive on the two inputs at any time, and if it is present on one input only it must be stored until the data arrives on the other input. Likewise, if a data item is required in more than one assertion from the same breakpoint, then it must be duplicated for each assertion. Accordingly, the assertions can be thought of as a dataflow graph, where the nodes perform functions and the arcs transmit tokens of data, captured at various instances of execution.

Using the example discussed in the previous section, Figure 4 shows a sample graph for the following assertion list:

assert *seq::A[0..49]:seq.c@100 = p1::B:par.c@100*

assert *seq::A[50..99]:seq.c@100 = p1::B:par.c@100*

assert *seq::A[0..9]:seq.c@150 = p1::B[0..9]:par.c@90*

assert *seq::A[50..59]:seq.c@150 = p2::B[0..9]:par.c@95*

The graph contains 4 compare nodes, each of which tests for a difference in their inputs. If there is a difference then this is reported through the node output. Figure 4 shows tokens moving along the input arcs, arriving at different times. Only when both tokens are present does the node fire. Tokens are produced as a result of breakpoints in each of the processes.

A few debuggers have used dataflow as a control mechanism in the past [15, 20]. In [20] this was concerned with using the breakpoint as an event trigger for a piece of debugger code. Likewise, TV [8] allows the user to write expressions which are executed when a breakpoint fires, and these can be considered like dataflow expressions. Our work extends the previous work by building complex tokens containing data from the processes being traced, and then using this data in the

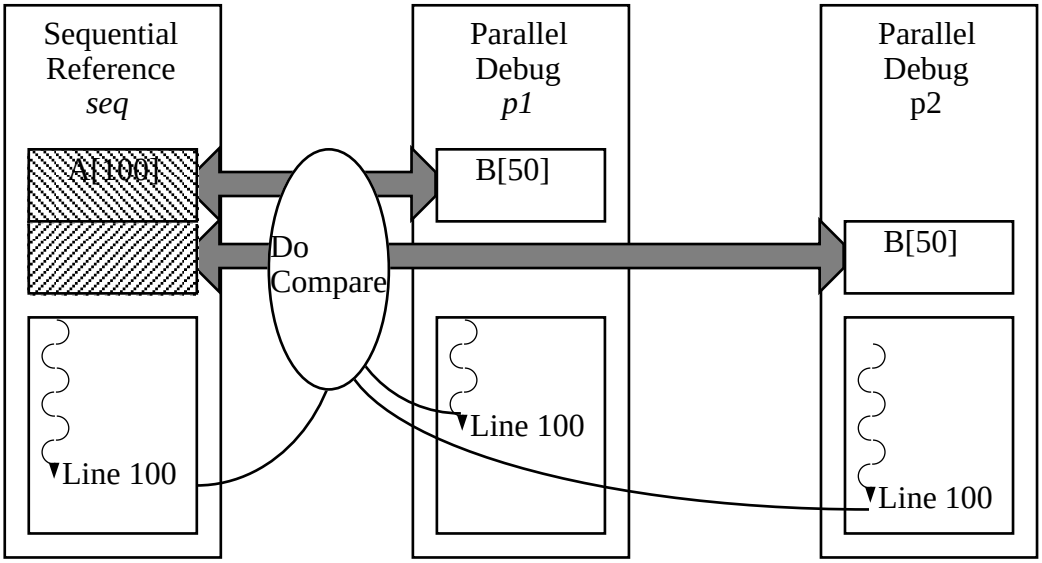


Figure 3 - Sequential Reference compared with Parallel version.

debug statements. The dataflow model makes it possible to detect when both operands of a compare have arrived.

In the next section we discuss some of the implementation considerations.

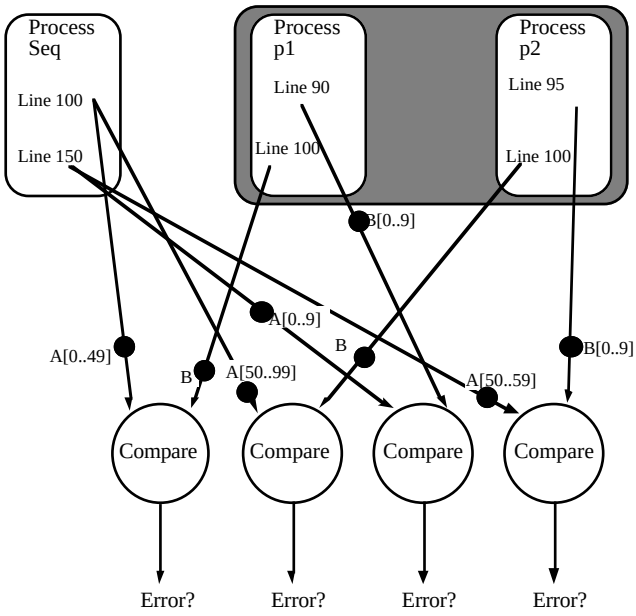


Figure 4 - Dataflow graph representing assertions

5. IMPLEMENTATION ISSUES

As discussed in the previous section, the most appropriate control mechanism to suit debugging in a parallel environment is one that utilities a dataflow model. When developing the implementation of such a model, we had to address a number of issues, including:

- the type of dataflow model to use;
- the types of nodes in the graph;
- what information will be carried by tokens in the graph;
- how the graph and tokens should be represented internally; and
- the representation of data in a heterogeneous environment.

These issues are addressed in more detail in the following sections.

5.1 Type of Dataflow Model

Dataflow architectures can be classified into two main categories: static and dynamic. In addition, there have been implementations of hybrid schemes [7, 10]. The primary difference between the static and dynamic models is the number of tokens that may be present on an input at one time. The static model allows only one, while in the dynamic model, many tokens may be present on the input to a node. Hybrid models are a combination of both strategies [2].

In our implementation, we chose to use a variation of the static model in which tokens form a FIFO queue on the input arcs. In this model tokens are consumed from the head of each queue. This makes it possible for us to restart a process immediately after the data has been extracted rather than wait for it to be processed. Each node also contains information which specifies which of the following conditions must be satisfied before the node will fire:

i. Fire only when tokens exist on all inputs;
ii. Fire when a token appears on any input;
iii. Fire once when the graph is loaded.

A node that fires will always consume its input tokens, but may not generate an output token. Nodes that have more than one output automatically replicate any output token on the different outputs. There are no limits on the number of inputs and outputs arcs for each node.

5.2 Graph Implementation

Internally, the dataflow graph is represented as a linked list of nodes. The data structure representing each node contains information about the number of inputs and outputs, the function that the node is to perform, and various other items.

Nodes that have tokens present on their inputs are linked into a list of *active* nodes. Execution of the graph consists of scanning this list and checking the inputs on each node. When the number of tokens matches the number required to fire the node (taking into account the class of the node), then the node is fired and the associated node function executed.

Since there are no tokens in the graph at the start, some mechanism must be used to generate some initial tokens. The generation of these initial tokens is handled by class (iii) nodes, which are primed with static information such as a program name.

5.3 Token Implementation

Creation and destruction of tokens involves the allocation and deallocation of memory resources, a potentially time consuming process. For efficiency, our implementation allocates a pool of tokens when it is first started, and can expand the pool dynamically if required. A bitmap structure is used to mark tokens in the pool that are currently free. The process of allocation then becomes simply a matter of finding the first free location in the bitmap, and then returning a reference to the corresponding token. Deallocation just requires the appropriate bit in the map to be set. Token duplication is managed by maintaining a reference count on the token.

Tokens are statically typed using a type tag. Nodes accept tokens of generic type by default. However, node functions can impose strict typing on the input tokens if required.

5.4 Data Representation

Central to the concept of relative debugging is the comparison of data from programs executing in a heterogenous environment. While it is relatively easy to perform conversions between simple data types such as integers and characters, it becomes more problematic for floating point values and complex data types such as arrays and structures. In order to overcome the problems inherent in this approach, it is important that an architecture independent format (AIF) for data representation is used.

All manipulation of data by the debugger is carried out in the AIF, using AIF support routines. Routines are also supplied to facilitate the conversion between the local machine dependent data format and the AIF.

Data represented in AIF consists of the components: the actual data formatted as a sequence of bytes of variable length; a format descriptor string (FDS) that consists of a variable length sequence of ASCII characters that describes the layout of the data; and the machine specific formatting information about the data.. The AIF support routines generally require both these components for operation.

The AIF makes some assumptions, including:

1) floating point values use IEEE implementation;
2) byte ordering is big endian (conversion carried out if required); and
3) all simple data types (including pointers) will fit into a maximum of 8 bytes.

The FDS can be parsed to determine the exact nature of the variable, including its type and structure. The syntax is sufficiently powerful describe both simple and complex types.

Below are some sample FDS strings for a few variable types:

Declaration	FDS	Meaning
`char c;`	`"cs"`	signed char (1-byte integer)
`int i;`	`"is4"`	4-byte signed integer
`float f[100];`	`"[r0..99is4]f4"`	array of 100 floating point numbers
`struct { int f1; float f4 };`	`"{8=f1@0#32:is4,f2@32#32:f4}"`	structure containing two fields, a 4-byte integer and a 4-byte floating point number

In these examples, a character is described by the string "cs", a four byte integer by the string "is4". A floating point array requires a description of both the index range, the type of the indexes and the type of the array elements, and is thus built from a range from 0 to 99, an index type of 4 byte integers and a base type of 4 byte floating point numbers. A structure with an integer and a floating point number is more complex. It contains a description of the size of each of the items (in bits) and the starting offset in the struct (also in bits), plus the base type of each of the items. Using this syntax we have been

able to represent all of the types found in major programming languages, including user defined types.

5.5 Debug Interface

A high level debugging interface is used to control the application programs and also to extract their state for comparison. This is provided by the Dynascope debug library [22, 23], which provides a client-server structures interface. Dynascope provides function to control the execution of programs on arbitrary machines, extract symbol table information and read and write variables in the process address space. More details of Dynascope can be found in [22] and [23].

6. A WORKED EXAMPLE

In this section we illustrate the use of dataflow graphs with a small, but realistic, example. Figure 5 shows a graph which performs the following assertion between two sequential programs, "prog1" and "prog2":

assert prog1::var1@prog1.c:8 =
prog2::var1@prog2.c:10

The graph begins by invoking both programs and planting breakpoints at line 8 in prog1 and line 10 in prog2. The *Set_Breakpoint* node returns a token which is then used to fire a *Continue* node. *Continue* starts the program running and waits for a breakpoint to be encountered, and then it emits a token. The output of the *Continue* node causes the variable "var1" to be read from both programs, and the data is then compared using a *Compare* node. If the data is equivalent, then the programs are both restarted, however, if an error is detected then the differences are reported to the user via a *Display* node.

A number of *Constant* nodes are used to inject literals into the graph. Some of these only fire once when the graph is started, whilst others are re-triggered repeatedly.

It is clear from this example that the system is not sensitive to the exact timing or order of breakpoint events. Further, additional assertions can be integrated easily into the graph, again without regard to timing issues.

7. CONCLUSION

In this paper we have described the design and implementation of an elegant and effective control mechanism for a relative debugger based on a dataflow model. Dataflow was chosen to provide support for the debugging of parallel codes relative to serial codes, where it can not be guaranteed information will arrive in a serial manner. The model also allows increased flexibility in the types of data comparisons that can be performed. The design is further enhanced by allowing additional functionality to be easily and quickly implemented and tested, if required.

We have implemented a prototype of the new control mechanism, which has been successfully trialed on a number of sequential programs. This has demonstrated that the new control mechanism can effectively replace the one used in the current implementation, and has addressed the primary issue of how the mechanism will interact with the application programs.

The prototype currently accepts dataflow graphs written in a simple linear syntax only. Plans are underway to examine alternate methods of specifying the dataflow graphs, including a linear declarative text form (similar to the current assertion syntax) and also using a visual graph editor. Possible additional enhancements include the ability to combine graph nodes into modules in order to reduce the overall complexity of the dataflow graphs.

In this paper we have not discussed the details of how parallel codes can be debugged using our relative paradigm, nor issues such as how the debugger will interact with parallel task managers. Rather, the paper has concentrated only on the core algorithms required for data gathering and comparison. We are currently in the process of designing nodes, in addition to the basic node set, that will provide support for parallel codes. We will report our experiences in addressing these issues, and present a more complete parallel relative debugger in a future paper.

Acknowledgments

This work is supported by the Australian Research Council. We wish to acknowledge our collaborators, Prof Larry Snyder, Dr Ian Foster and Professor Geoff Dromey who proof read drafts of this paper. We also thank Mr Borut Lesjak who designed and implemented the Architecture Independent Format (AIF) used in the implementation, plus sections of the Dynascope debugging system.

References

[1] Abramson D., Foster, I., Michalakes, J. and Sosic R., "Relative Debugging and its Application to the Development of Large Numerical Models", IEEE Supercomputing 1995, San Diego, December 95.

[2] Abramson, D and Egan, G, "The RMIT Dataflow Computer: A Hybrid Architecture", The Computer Journal, Vol 33, No 3, June 1990, pp 230 - 240.

[3] Abramson, D.A. and Sosic, R. "A Debugging Tool for Software Evolution", CASE-95, 7th International Workshop on Computer-Aided Software Engineering, Toronto, Ontario, Canada, July 1995, pp 206 - 214.

[4] Abramson, D.A. and Sosic, R. "Relative Debugging using Multiple Program Versions", Key Note Address, 8th Int. Symp. on Languages for Intensional Programming , Sydney, 3-5th May,

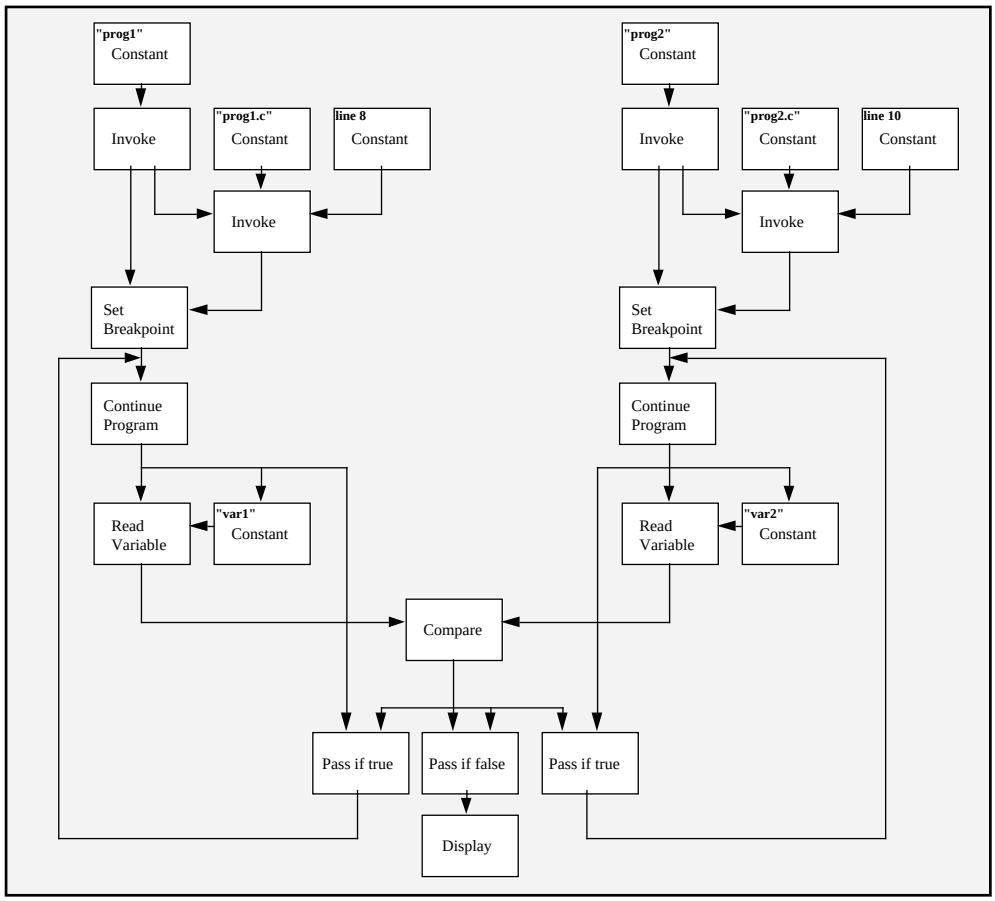


Figure 5 - Sample Dataflow graph

1995. In Intensional Programming I, World Scientific, ISBN 981 - 02 - 2400 - 1.

[5] Abramson. D.A., and Sosic,. R. "Method for Testing and Debugging Computer Programs", Australian Provisional Patent, PM5196/94.

[6] Aral, Z., Gertner, I. and Schaffer., G. "Efficient debugging primitives for multiprocessors", Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, pages 87-95. ACM, 1989.

[7] Arvind, Bic, L. and Ungerer, T. "Evolution of Data-flow Computers", Chapter 1, "Advanced Topics in Dataflow Computing", Prentice Hall, 1991.

[8] BBN Corporation, "The Total View Debugger Reference Manual"

[9] Cheng, D. and Hood, R. "A Portable Debugger for Parallel and Distributed Programs", Proceedings of Supercomputing 94, Dec 94.

[10] Dennis, J. "The Evolution of "Static" Data-flow Architectures", Chapter 2, "Advanced Topics in Dataflow Computing", Prentice Hall, 1991.

[11] Elshoff. I. J. P., "A distributed debugger for Amoeba", Proceedings SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, pages 1-10. ACM, 1988.

[12] Griffin, J. H, Wasserman, H. J. and . McGavran, L. P. "A debugger for parallel processes", Software-Practice and Experience, 18(12):1179-1190, December, 1988.

[13] LeBlanc, T. J. and Mellow-Crummey, J. M., "Debugging parallel programs with instant replay", IEEE Transactions on Computers, 36(4):471-482, April 1987.

[14] Lin, C. and Snyder, L. , ZPL: an array sublanguage. In U. Banerjee, D. Gelernter, A. Nicolau, And D. Padua (Eds.) Languages and

Compilers for Parallel Computing, Springer-Verlag, pp. 96-114, 1993.

[15] Lumpp, J, Casavant, T., Siegel, H.J., and Marinescu, D. "Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems", Proceedings of 10th Intl. Conference on Distributed Computing Systems, IEEE, 1990, pp 476 - 483.

[16] McDowell, C. E. and Helmbold, D. P. , "Debugging concurrent programs", ACM Computing Surveys, 21(4):593-622, December 1989.

[17] McRae G.J., Russell A.G. and Harley R.A., 1992. CIT photochemical airshed model -users manuals, Carnegie Mellon University, Pittsburgh, PA and California Institute of Technology, Pasadena, CA.

[18] Miller, B. P.and Choi, J.-D., "A mechanism for efficient debugging of parallel programs", Proc. SIGPLAN'88 Conference on Programming Language Design and Implementation, pages 135-144. ACM, 1988.

[19] Moher, T. G. "PROVIDE: A process visualization and debugging environment", IEEE Transactions on Software Engineering, 14(6):849-857, June 1988.

[20] Olsson, R. A. , Crawford, R. H. and Ho W. W. . "A dataflow approach to event-based debugging", Software-Practice and Experience, 21(2):209-229, February 1991.

[21] Ramsey, N. and Hanson, D.R. "A retargetable debugger"., Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation, pages 22-31. ACM, 1992.

[22] Sosic, R. "A Procedural Interface for Program Directing", Software Practice and Experience, Vol 25, No 7, July, 1994, pp 767 - 787.

[23] Sosic, R. "Design and Implementation of Dynascope, a Directing Platform for Compiled Programs", Vol 8, No 2, Spring, 1995, pp 107 - 134.

[24] Satterthwaite. E. " Debugging tools for high level languages", Software-Practice and Experience, 2(3):197-217, July-September 1972.

[25] Stallman, R. M., GDB Manual, Free Software Foundation, Cambridge, MA.

Parallel Relative Debugging for Distributed Memory Applications: A Case Study

Gregory R. Watson and David Abramson

School of Computer Science and Software Engineering
Monash University
Wellington Road, Clayton, VIC 3168, AUSTRALIA

Abstract

Relative debugging is a technique that addresses the problem of debugging programs that have been developed using evolutionary software techniques. Recent developments allow relative debugging to be used on programs that have been ported from serial to parallel architectures or between different parallel architectures. Such programs may change significantly in the porting process and this raises many issues for the debugging methodology. This paper examines the use of relative debugging on a distributed memory application in which errors were introduced when the code was ported from a sequential to a parallel architecture. Our debugger, GUARD-2000, is used to compare key data structures between the two codes even though the parallel data structure has undergone significant reorganisation when mapped onto a distributed memory platform. We show how this technique can quickly and accurately pinpoint the source of errors in the parallel code.

INDEX TERMS: parallel programming tools, relative debugging, distributed memory, porting.

1 Introduction

Relative debugging is a technique that allows a user to compare data between two executing programs [17]. It was devised to aid the testing and debugging of programs that are either modified in some way or are ported to other computer platforms. Traditional debuggers force the programmer to understand the expected state and internal operation of a program. In contrast relative debugging makes it possible to trace errors by comparing the contents of data structures between programs at run time. In this way the programmer is less concerned with the actual state of the program and more concerned with finding when and where differences between the old and new codes occur. Relative debugging operates by allowing the user to define a series of *assertions* between a *reference* program and a *suspect* program. These assertions specify key data structures that must be equivalent at specific locations in the codes. When the programs are executed the debugger is able to use the assertion information to automatically compare the data structures and report any differences to the user.

A number of case studies over the years have demonstrated the significant power of relative debugging as a practical technique for locating errors experimentally [5][18]. However, parallel programs provide additional challenges in debugging [8]. Importantly, in many cases they are converted from existing sequential codes rather than being rewritten from scratch. We have argued that relative debugging is an important tool aimed at simplifying the porting of sequential programs to parallel platforms because it recognises the existence of a working sequential version that can be used as a comparison point. Until recently we have only been able to demonstrate this for sequential programs and a restricted set of parallel codes. This was because our early implementations of the debugger did not have sufficient expressive power to deal with distribution of data across a parallel machine, nor the changes that are necessary in data structure when the program is ported.

In this paper we present a case study that illustrates the application of relative debugging *when a program is ported from a sequential language to a parallel version running on a distributed memory platform.* The example illustrates a novel technique that is required in such an environment, namely the ability to

describe the way the data is distributed across the memory of the parallel machine. The paper gives a very brief description of relative debugging, and its implementation called GUARD. It then describes the application under consideration which is known as the Shallow Water Equations [16]. This is followed by a discussion of how GUARD facilitates data comparison across distributed memory machines.

2 Relative Debugging

Unlike other conventional parallel debuggers [6][12][13] a relative debugger provides the ability to dynamically compare data between two executing programs regardless of their location and configuration. In our implementation of a relative debugger, called GUARD, data comparisons can be performed either using an *imperative* scheme or a *declarative* scheme. Imperative comparisons can be performed explicitly by the user when two programs under the control of the debugger are stopped at breakpoints. The imperative `compare` command behaves like a conventional debugger data display command, however it names data structures in two processes instead of one. For example the following command compares the data from array `A` in `$proc1` with array `B` in `$proc2`:

```
compare $proc1::A = $proc2::B
```

If the arrays differ then the differences are reported. The compare command requires a fairly high degree of interaction by the user because the two programs must be manually executed and halted before the data can be compared.

An alternative way of comparing data is to use the declarative `assert` command. This command takes a pair of data structure names, process and file names and line numbers and compares the contents of the data structures only when the processes have reached their respective line numbers. The following command compares the data from array `A` in `$proc1` with array `B` in `$proc2` when `$proc1` reaches line 123 and `$proc2` reaches line 456:

```
assert $proc1::A@"file1.c":123 =
          $proc2::B@"file2.f":456
```

When the user issues an assert command the debugger compiles it into a dataflow graph which is then executed under the control of an interpreter. Dataflow nodes only fire when the programs reach breakpoints and the comparison is performed only when both data structures are available.

The relative debugging methodology involves placing assertions at key points in the two programs until a difference in data is detected. At this point the user examines the control and data flow of the programs and new assertions are written with the aim of refining the region that is in error. This process means that it is possible to reduce the region under consideration to a small code fragment very quickly because the region is divided on each pass. Conventional debug methodology and tools can be used to finally isolate the error. It is beyond the scope of this paper to describe in detail how relative debugging is implemented however the scheme is fully described in [3]. We refer the reader to a number of other papers that provide a thorough introduction to relative debugging and how the technique can be used to find the location of errors [2][4].

The assert and compare commands provide a conceptually simple mechanism for comparing data between two sequential programs. However a key goal of relative debugging is to extend this mechanism to encompass the comparison of data between related sequential and parallel programs. In this situation the comparison process is complicated by the decomposition that is normally applied to sequential data structures when a code is ported to a parallel architecture. To address this issue GUARD provides a `map` function which is used to define the data decomposition that has been employed. Rather than just supporting standard decompositions such as BLOCK and CYCLIC [7] as found in languages like HPF [10] we allow the user to write an arbitrary arithmetic expression that dictates the conversion of indexes from one space to another. Once the mapping has been specified it can be used in an assert command to provide the debugger with the necessary information needed to perform the comparison operation. For example if the array `B` above has been distributed using a block-cyclic decomposition defined by the map `bcyc` then the assertion might be written as:

```
assert $proc1::A@"file1.c":123 =
       bcyc($proc2::B@"file2.f":456)
```

Once a data distribution mapping has been defined the user no longer needs to be concerned with the decomposition issues and can again concentrate on locating the source of the error. A complete description of parallel relative debugging is given in [19]. The utility of this technique is demonstrated in the following case study which examines the use of relative debugging to locate errors in a distributed memory parallel code.

3 Distributed Memory Case Study

The program used in this case study is a simple numerical model of the shallow water equations [15][9]. Abramson, et al. ported the shallow water equations to a number of languages and architectures [1]. Both the sequential and parallel versions are written in the C language. The parallel program is a hand-coded distributed memory implementation using the MPI parallel architecture [14]. The application is interesting because it represents a simplified model of the dynamics of a real weather model [11] and it captures features of the computation such as the communication and data distribution requirements. The equations are discretized across a regular grid and a finite difference formulation is used to approximate the partial derivatives. The program time steps from one iteration to another updating the key variables on each step. Figure 1 shows the basic program call tree.

```
main()
{
  int ncyc;

  initialise(u,v,p,psi,di,dj);
  ...
  for (ncyc = 0; ncyc < itmax; ncyc++) {
    /* Calculate cu, cv, z and h*/
    calcuvzh(p,u,v,cu,cv,z,h,fsdx,fsdy);
    ...
    /* Calculate time tendencies */
    timetend(dudt,dvdt,dpdt,z,cv,cu,h);
    ...
    /* Calculate new values */
    tstep(u,v,p,dudt,dvdt,dpdt,first,tdt);
    ...
    if ( first ) {
       /* leapfrog */
      tdt = tdt+tdt;
      first = 0;
    }
  }  /* End of time step loop */
  ...
}
```

Figure 1: "shallow" program call tree

The distributed memory code uses a master/slave arrangement, where the master maintains primary copies of the key data structures. Each slave is sent an entire copy of the data structures, but only performs computations on a portion determined by slicing the outer loop using a traditional BLOCK decomposition [7]. Before and after a calculation, each slave synchronises the edges of the data structures with its immediate neighbours. At the end of the time step loop the slave data is copied back to the master process. This arrangement is shown in Figure 2.

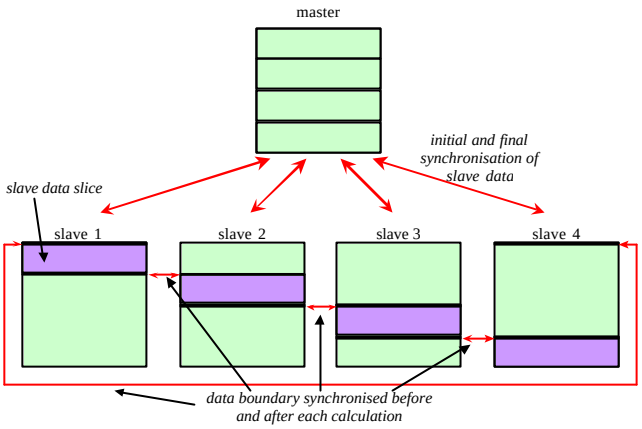


Figure 2: Data decomposition and boundary synchronisation for "shallow"

Both codes use a number of key variables in the computation of the shallow water equations. These include the zonal wind (`u`), meridonal wind (`v`), pressure (`p`), potential enstrophy (`z`) and quantity (`h`) as well as mass weighted zonal wind (`cu`) and meridonal wind (`cv`). In addition the codes maintain time tendency variables for the zonal wind (`dudt`), meridonal wind (`dvdt`) and pressure (`dpdt`). Both codes compute new values for `cu`, `cv`, `z` and `h` in the routine `calcuvzh()` and perform the time tendency calculations in the routine `timetend()`.

In order to begin debugging the "shallow" code, a map that describes the data decomposition technique used by the distributed memory code was defined. One major difference between the codes is the index ordering which has been swapped in the translation from sequential to parallel. The map was also defined to account for this index permutation.

```
map shallow(P::A)
  define index(i,a,u,l) = a[1-i]
  define proc(i,a,u,l) =
             a[1] * $procs / $rows
end
```

The first line of the map indicates that the ith index of the array should be swapped with the $(1-i)^{th}$ index. This effectively swaps the indices of a two-dimensional array. The second line of the map specifies which process holds each element of the array. In this case multiplying the first index value by the number of processes and dividing by the number of rows in the array determines the process number. i.e. rows 0 through `rows/procs-1` are located on process 0, `rows/procs` through `2*rows/procs-1` are located on process 1, and so on. The full mapping syntax and semantics can be found in [19].

For this case study a model size of 32×32 and a run length of 950 time steps was chosen. The distributed memory code was initially run on two processors. The codes are designed to report a number of key values every 50 time steps so it was simple to determine if the codes were working correctly or not. Figures 3a and 3b show the output from the two versions of the code after 950 time steps and in which significant errors can be clearly seen in all four values.

```
Cycle number 950
Model time in days 0.99
  Potential energy 6816.106
  Kinetic Energy   41183.156
  Pot. Enstrophy   2.006166e-27
  Total Energy     47999.262
```

(a) Sequential

```
Cycle number 950
Model time in days 0.99
  Potential energy 6790.644
  Kinetic Energy   41219.078
  Pot. Enstrophy   6.152948e-17
  Total Energy     48009.723
```

(b) Distributed Memory

Figure 3: Output from the "shallow" code

3.1 Error 1: Incorrect Index Value

The shallow water mapping was used to define a series of assertions to compare the values of the key data structures `u`, `v` and `p` between the sequential and parallel codes just prior to entering the main time step loop. This tests the initialisation code in the two programs, and ensures that the variables are correct before any calculations are undertaken. The error limits were set to an initial estimate of the likely error range. The commands used were as follows (assuming `$c` represents the sequential program and `$p` the parallel program):

```
set error 1.0e-20 1.0e-1
assert $c::u@"cshallow.c":61 =
       shallow($p::u@"main.c":405)
assert $c::p@"cshallow.c":61 =
       shallow($p::p@"main.c":405)
assert $c::v@"cshallow.c":61 =
       shallow($p::v@"main.c":405)
```

Execution of these assertions showed that `u`, `p` and `v` are correct prior to entering the time step loop. This confirms that the errors must be being introduced by calculations in this loop.

The next step was to examine the result of the computations performed in `calcuvzh()`. This was done by defining assertions to compare the values of the variables `cu`, `cv`, `h` and `z` as follows:

```
assert $c::cu@"cshallow.c":71 =
       shallow($p::cu@"main.c":430)
assert $c::cv@"cshallow.c":71 =
       shallow($p::cv@"main.c":430)
assert $c::h@"cshallow.c":71 =
       shallow($p::h@"main.c":430)
assert $c::z@"cshallow.c":71 =
       shallow($p::z@"main.c":430)
```

After first call to `calcuvzh()` differences were observed in the variable `cv`. These differences are shown in Figure 4a. To visualise the differences in the arrays we use a two-dimensional representation where each pixel shows the value of the difference at the corresponding location in the arrays. The relative magnitude of the differences are mapped onto a colour table with white representing no differences and black the largest difference. Using this technique the differences observed in the `cv` arrays suggested that the error was related to a boundary calculation by each processor.

To confirm this suspicion the codes were re-run with the distributed memory code executing on four processors rather than the initial two. Errors were again observed in the variable `cv` however this time the differences appeared as shown in Figure 4b. This confirmed that `cv` was affected by a boundary issue that depended on number of processes. Since the error was in the

boundary column the corresponding boundary code was examined for coding defects. Inspection of the `calcuvzh()` code (12 lines) lead to the discovery of an incorrect array index in the calculation of cv.

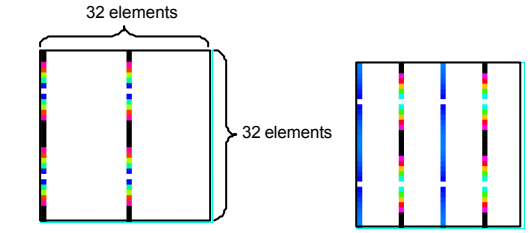


Figure 4: Boundary errors observed in the `cv` variable

Relative debugging allowed the rapid identification of this error by observing the changes resulting from running on different numbers of processors, and irrespective of the fact that the array indexes are permuted.

3.2 Error 2: Wrong Array Element

After correcting the index value and re-running the programs using the same assertions errors were still observed in the `cv` variable. The characteristics of the differences had altered significantly however and now seemed to indicate a periodic error in the `cv` calculation. The new differences are shown in Figure 5.

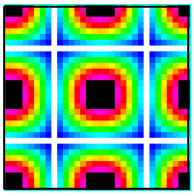


Figure 5: Apparent periodic errors in the `cv` variable

Further inspection of the `calcuvzh()` revealed a second error in the calculation of `cv`. In this case the wrong element of the array `v` was used in the calculation of `cv`.

This error was interesting because it occurred in the same statement as the previous error, but was missed in the original code inspection. Because the nature of the error in a single variable changed from a boundary problem to a periodic problem the source of the problem was quickly identified.

3.3 Error 3: Wrong Sign

After ensuring that both errors had been corrected in the parallel code the programs were rerun to check that the `cv` variable was now correct. The assertions showed that all four calculated variables, `cu`, `cv`, `h` and `z` were correct after first iteration, however on subsequent iterations errors were observed accumulating in the `cu` variable. Figure 6 shows an example of these errors.

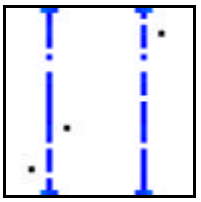


Figure 6: Errors observed accumulating in the `cu` variable after first time step

Inspection of the code showed that the calculation of `cu` is derived from the values of `p` and `u`. Since all values of `cu` are correct on first iteration but incorrect on second and subsequent iterations, this implies that there must be in error in the computation of `p` and `u` (which were shown to be correct immediately after initialisation).

The following assertions were defined in order to check the values of `p` and `u` at the end of the time step loop:

```
assert $c::u@"cshallow.c":85 =
       shallow($p::u@"main.c":449)
assert $c::p@"cshallow.c":85 =
       shallow($p::p@"main.c":449)
```

When the codes were again run errors were immediately reported in `u`. These are shown in Figure 7.

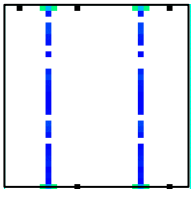


Figure 7: Errors observed in `u`, which is used to derive `cu`

At this point it was clear that errors were being introduced into the variable `u` but these could have resulted from either the calculation of

`u` itself in `tstep()`or in the time tendency variable `dudt` which is calculated in `timetend()`. To verify which routine was the source of the errors it was necessary to rerun and check the value of `dudt` immediately after the call to `timetend()`. Figure 8 shows the differences visible in dudt when this was done.

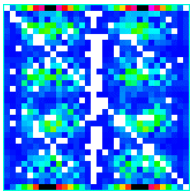


Figure 8: Errors resulting from wrong sign in calculation of `dudt`

It was now known with reasonable certainty that the errors were occurring in the `timetend()` routine. At this point a visual inspection of the 15 lines of code in `timetend()` was used to locate the source of the error. In this case the error turned out to be a wrong sign used in the calculation of `dudt`.

This error highlights the use of relative debugging to quickly reduce the possible location of an error to a small region of code. Once this has been done, simple code inspection will usually pinpoint the incorrect statement immediately even when the actual error is very minor as in this case.

4 Conclusion

In this paper we show how our relative debugger, GUARD, can be used to locate errors in a code that has been ported from a sequential architecture to a distributed memory architecture. This work is significant because it is the first time that relative debugging has been used to locate errors in a hand-coded distributed memory code running on multiple processes. The case study shows that the debugger:

- easily handles significant changes to key data structures that are introduced during the porting process; and
- can accurately locate the source of errors in only a few steps using an iterative refinement process.

One criticism of the mechanism used by the debugger to describe data decomposition is that the user may write an incorrect transformation and thus the debugger will indicate that the structures are incorrect even though they are actually the same. In addition the mapping language itself is inherently fragile. That is, minor coding errors can lead to major changes in the final specification. The consequences of such mapping errors are that it may become difficult or impossible to locate the original program errors or that 'spurious' errors may be introduced. We propose a number of solutions to overcome these shortcomings. First we believe that the use of pre-packaged transformations will reduce the occurrence of this situation. It should be possible to provide transformations to replicate most normal decomposition techniques such as block and block-cyclic. Map creation could also be facilitated through the use of either a semi- or fully automated analysis system. A semi-automated system could provide a graphical environment for specifying maps and allowing the user to model and test the decomposition or transformation interactively. A fully automated system could perform code analysis on the serial and parallel codes to determine the required mapping.

In conclusion it is worth indicating that relative debugging is of limited value when timing and other non-deterministic errors occur in parallel codes. The technique is very effective at locating a region that is in error and it will flag errors between a working program and an erroneous one. However, because the debugger disturbs the timing of the program any errors it isolates may vary from one execution to another if non-deterministic behaviour is present.

Acknowledgements

This work has been supported by the Australian Research Council under the Large Grant Scheme and by Microsoft Corporation.

References

[1] Abramson, D., Dix, M. and Whiting, P., "A Study of the Shallow Water Equations on *Various* Parallel Architectures", *14th Australian Computer Science Conference*, pp. 06-1 – 06-12, Sydney, 1991.

[2] Abramson, D., Sosic, R., "A Debugging Tool for Software Evolution", CASE-95, *7th International Workshop on Computer-Aided Software Engineering*, pp 206 – 214, Toronto, Canada, July 1995.

[3] Abramson, D., *Sosic*, R., Watson, G., "Implementation Techniques for a Parallel Relative Debugger", *Proceedings of PACT '96*, Boston, October 1996.

[4] Abramson, D., Foster, I., Michalakes, J., Sosic, R., "Relative Debugging: A New Methodology for Debugging Scientific *Applications*", Communications of the ACM, Vol. 39, No. 11, pp 69 – 77, November 1996.

[5] Abramson D., Foster, I., Michalakes, J., Sosic R., "Relative Debugging and its Application to the Development of Large Numerical Models", *Proceedings of IEEE Supercomputing 1995*, San Diego, December 1995.

[6] Cheng, D., Hood, R., "A Portable Debugger for Parallel and Distributed Programs", *Proceedings of Supercomputing '94*, pp. 723 – 732, November 1994.

[7] Foster, I., Designing and Building Parallel Program: Concepts and Tools for Parallel Software Engineering, Addison-Wesley Publishing Co., 1995.

[8] Gait, J., "A Probe Effect in Concurrent Programs", *Software Practice and Experience*, Vol. 16, No. 3, pp. 225 – 233, 1986.

[9] Haltiner, G.J. and Williams, R.T., "Numerical Prediction and Dynamic Meteorology", Wiley, 1980.

[10] High Performance Fortran Forum, "High Performance Fortran Language Specification", *CRPC-TR92225*, Version 2.0, Center for Research on Parallel Computation, Rice University, , January 1997.

[11] Hoffmann, G.-R. and Maretis, D.K., *The Dawn of Massively Parallel Multiprocessing in Meteorology*, Springer-Verlag, 1990.

[12] Kacsuck, P., Cunha, J., Dózsa, G., Jourenço, J., "A Graphical Development and Debugging Environment for Parallel Programs", *Parallel Computing*, Vol. 22, No. 13, pp. 1747 – 1770, 1997.

[13] May, J., Berman, F., "Panorama: A Portable Extensible Parallel Debugger", *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 96 – 106, San Diego, May 1993.

[14] Message Passing Interface Forum, *MPI: A message-passing interface standard*, International Journal of Supercomputer Applications, Vol. 8, No. 3/4, pp 165 – 414, 1994.

[15] Pedlosky, J., *Geophysical Fluid Dynamics*, Springer-Verlag, 1979.

[16] Sadourny, R., "The Dynamics of Finite-Difference Models of the Shallow Water Equations", *Journal of Atmospheric Science*, Vol. 32, pp. 680 – 689, 1975.

[17] Sosic, R., Abramson, D., "Guard: A Relative Debugger", Software – Practice and Experience, Vol. 27, No. 2, pp 185 – 206, February 1997.

[18] Watson, G. and Abramson, D. "Relative Debugging For Data Parallel Programs: A ZPL Case Study", IEEE Concurrency, Vol 8, No 4, pp 42 – 52, October 2000.

[19] Watson, G. R., *The Design and Implementation of a Parallel Relative* Debugger, PhD Thesis, Monash University, Melbourne, October 2000.

Greg Watson and David Abramson
Monash University, Australia

Relative debugging is a powerful paradigm that lets us locate errors in programs that result from porting or rewriting code. The authors describe their experience using relative debugging to compare a program written in a sequential language with one that was ported to the data-parallel language ZPL.

Relative Debugging for Data-Parallel Programs: A ZPL Case Study

In this article, we describe Guard99, a new implementation of relative debugging for parallel platforms that lets a programmer locate errors by observing the divergence in key data structures as two programs are simultaneously executed. The implementation uses a client–server, machine-independent architecture, so it can debug programs that execute on different platforms. By employing an architecture-independent implementation layer, the debugger is able to hide variations in data representation on these different platforms from the user.

The case study we report represents the first time a relative debugger has been used to locate errors by comparing one program written in a data-parallel language to another written in a sequential language (for more information on relative debugging, see the related sidebar). The data-parallel language ZPL,[1] developed at the University of Washington, incorporates special linguistic constructs that let a programmer specify parallel data structures and how to perform the computation in parallel. ZPL is a high-level machine-independent parallel programming language that has been ported to a wide variety of platforms.[2] Our decision to use it in the case study was arbitrary but logical, because we had access to the language designers and an actual implementation. The techniques described here apply equally well to other data-parallel languages, and we designed the debugger implementation so that adding support for additional languages is a straightforward process.

Guard99 found errors previously not located in a sample ZPL program—the programmers assumed the program was operating correctly even though there were minor numeric differences. Guard99 showed that some of these differences could be attributed to variations in the execution of sequential and parallel versions of ZPL. The work is particularly important because a programmer not fluent in ZPL or the underlying application was able to debug the ZPL code. The errors were located quickly and efficiently.

ZPL

ZPL is a data parallel array-based language[3] that supports the array as a fundamental data type. It also provides various features that let programmers generalize algorithms using array semantics. Although it is primarily designed as a data-parallel language, ZPL executes on both sequential and parallel architectures. It is also an implicitly parallel language, as the programmer does not need to explicitly specify how parallel computations are to take place; rather, the compiler determines the distribution of data automatically. Parallelism is derived from the

1092-3063/00/$10.00 © 2000 IEEE

Relative debugging

Relative debugging is a high-level technique that lets us compare data in a *reference* program to that of a *suspect* program.[1,2] Often, programs derived from a reference program, perhaps through porting or some other form of an evolutionary software process, suffer from introduced errors. Relative debugging provides a powerful technique for locating porting errors quickly. Various case studies reporting the results of using relative debugging have been published.[1,3–4] The concept of relative debugging is both language and machine independent. It lets users compare data structures without concern for the implementation, so they can focus on the cause of the errors rather than on the implementation details.

Several versions of relative debugging have been built. The first one, Guard95, only supported programs running on sequential platforms.[2] Guard95 included some limited machine independence, but this was mostly restricted to supporting machines with different byte ordering. The current implementation, Guard99, provides support for parallel computer systems using an enhanced dataflow mechanism,[5] and it also includes a machine-independent layer. It lets us compare data when the compiled structures differ and when the machine architectures use different byte orderings or word sizes. For example, it can compare real numbers in different formats and sizes, as well as characters that are represented by different collating sequences.

In our case study, arrays are implemented quite differently in C and ZPL, as a ZPL array can be physically distributed across a number of processes, relying on complex data structures to manage the parallel implementation issues. Guard99 lets the user maintain this viewpoint by ensuring that these differences are hidden. The ZPL runtime library and Guard99 transparently manipulate parallel data structures to allow the comparisons to be performed (we discuss the debugger architecture that supports these features in the main text).

To the user, a relative debugger seems like any traditional debugger except that it also provides additional commands for comparing data from different processes. The debugger can control more than one process simultaneously, so when the processes halt at breakpoints, data comparison can be performed using the `compare` statement.

This technique is known as an imperative comparison and works well for simple cases. An alternative declarative comparison technique is also provided to allow the user to define a set of criteria for the correct operation of the suspect program a priori. These criteria are defined using a series of assertion commands.

When performing comparisons, errors might be incorrectly attributed to differences in the precision of the program variables or other minor numeric factors. To avoid this, Guard99 lets the user specify a tolerance value. Variables are considered equivalent when the result of a comparison is below this value. Two different types of tolerance are supported: absolute and relative. For absolute tolerance, the magnitude of the difference between the variables is compared directly with the tolerance value. So, for some tolerance ε, the error is ignored if $|v_1 - v_2| < \varepsilon$. In situations where the values are very small but the differences still constitute a significant error, relative tolerance can be used. Here, the difference is first divided by the maximum of the two variables before being compared to the tolerance. In this case, any errors will be ignored if

$$\frac{|v_1 - v_2|}{\max(|v_1|, |v_2|)} < \varepsilon \quad .$$

Guard99 provides three methods for visualizing comparison results: a simple text format, hierarchical data format (HDF), and architecture-independent format (AIF).[5] The text format is generally used only for quickly visualizing simple data structures, as it becomes much too unwieldy for large amounts of data. HDF and AIF are intermediate formats that can be used as input into a visualization package to generate 2D and 3D representations of the data. HDF is limited to multidimensional scientific data sets but is currently the de facto standard data format. AIF allows arbitrarily complex data structures to be represented, and we discuss the format in more detail in the ZPL section of the main text because it underpins the debugger's machine and language independence.

Some evidence suggests that visualizing comparisons, particularly using 2D and 3D representations, provides the user with a means of characterizing patterns of differences. In a previous study, a time-series isosurface representation of the error was used to identify independent errors in a mesoscale weather model. In particular, the isosurface's structure lets us identify errors in different code sections that were responsible for various physical processes. For example, an error in the physics on the planetary boundary layer was identified through an isosurface that was visible at the bottom of the 3D space. Another, different error in the long-wave radiation physics code was visible in the top of the atmosphere. This and other studies are discussed in more detail elsewhere.[4] Some differences in our case study show characteristic periodic behavior, suggesting problems involving trigonometric operations. In all these cases, we can make generalizations about the nature of the patterns, but until further research is conducted in this area, these are currently limited to being used for insight when making deductions about the nature of the errors.

References

1. D. Abramson and R. Sosic, "A Debugging and Testing Tool for Supporting Software Evolution," *Automated Software Eng.*, Vol. 3, Nos. 3–4, Aug. 1996, pp. 369–390.
2. R. Sosic and D. Abramson, "Guard: A Relative Debugger," *Software—Practice and Experience*, Vol. 27, No. 2, Feb. 1997, pp. 185–206.
3. D. Abramson and R. Sosic, "A Debugging Tool for Software Evolution," *CASE-95, Seventh Int'l Workshop Computer-Aided Software Eng., Proc. 7th Int'l Workshop Computer-Aided Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1995, pp 206–214.
4. D. Abramson et al., "Relative Debugging: A New Methodology for Debugging Scientific Applications," *Comm. ACM*, Vol. 39, No. 11, Nov. 1996, pp. 69–77.
5. D. Abramson, R. Sosic, and G. Watson, "Implementation Techniques for a Parallel Relative Debugger," *Proc. PACT '96*, IEEE Computer Soc. Press, Los Alamitos, Calif, 1996.

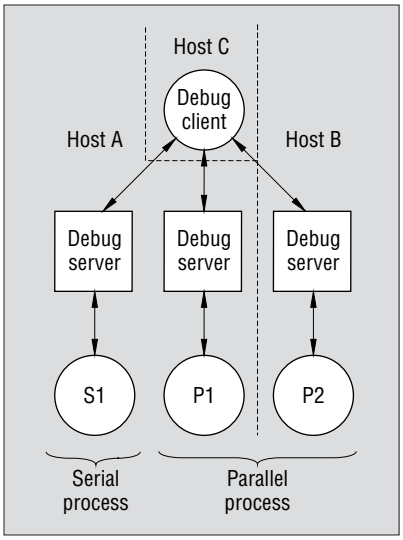


Figure 1. The client–server architecture Guard99 uses.

semantics of the array operations, so there are no parallel directives or mechanisms for explicit message passing. Programmers see a single address space and can use traditional sequential programming semantics for software development.

ZPL provides most attributes of traditional sequential programming languages. In addition, it supplies a number of new features to support the array semantics. *Regions* specify either a set of indices that define an array's bounds or a section of the array over which computations are performed. For example, the following declarations define a 10×10 region and an array containing 100 elements:

```
region R = [1..10, 1..10];
var A: [R] double;
```

Prefixing a statement with a region specifies that all operations on arrays of the region's same rank are to be performed for the indices defined by the region. The statement

```
[R] A := B + C
```

results in the sum of elements of `B` and `C` that are within the region `R` being stored in the corresponding elements of `A`. Any elements of `A` that are outside the region are unchanged by the statement.

Vector constants are denoted in ZPL using *directions*, which let a programmer specify relative positions that can perform transformations on regions. A typical declaration might be

```
direction north = [-1,0];
          east = [0,1];
```

There are a number of special operators for transforming or referencing regions. The `of` operator takes a direction and region as its operands and defines a new boundary region that is adjacent to the base region. The `@` operator, when applied to an array, translates the array's region by a specified direction, then references those elements of the array. Thus, to refer to elements of `B` obtained by adding the direction `north` to each index tuple in `R`, we'd use the following statement:

```
[R] A := B@north;
```

Periodic and mirrored boundary conditions are supported using the `wrap` and `reflect` operators. The `wrap` operator causes values on an array's boundary to be copied from the "opposite" side of the array, while the `reflect` operator results in values on the boundary of an array being copied from elements adjacent to the boundary. `Wrap` and `reflect` are typically used as follows:

```
[north of R] wrap A;
[east of R] reflect B;
```

Traditional reduce and scan semantics that apply a function accumulatively over an array are also supported. The `reduce` operator produces a scalar value representing the accumulated result of the function applied to each element in the array. The result of a `scan` operation is an array in which the *i*th element is the accumulation of the function applied to the first *i* elements of the array, in row-major order. In the following statements

```
Sum := +<<A; Plus := +||A;
```

the sum of all the elements of `A` is stored in the scalar `Sum`, and each element in the array `Plus` contains the sum of the preceding elements in each row.

ZPL achieves a high degree of machine independence by compiling to SPMD code for an abstract parallel machine in which the processes are arranged in a 2D mesh. The ZPL compiler developed at the University of Washington generates ANSI C code that is postcompiled using a native C compiler on the target system. This C code interfaces to libraries that emulate the abstract parallel machine on the target architecture and provide a well-defined interface to architecture-specific features. In the process, the logical process mesh is mapped onto the target processor topology.

Architecture details

A relative debugger's architecture has more functional requirements than traditional sequential and parallel debuggers. In addition to the need to control two or more processes simultaneously, the processes being debugged might also run on physically distributed systems, possibly employing different machine architectures. Managing assertions requires simultaneously controlling multiple processes and extracting data at arbitrary times in the execution life of the process. Support for data-parallel languages requires the debugger to interpret the parallel data structures that the language runtime system uses.

Guard99 addresses these requirements through a variety of different mechanisms. A client–server architecture controls multiple processes on distributed systems and architectural differences are handled by providing a machine-independent data representation that is used for all data manipulation activities. A dataflow compiler translates user-specified assertions into a graph that a dataflow engine can later execute. Using a dataflow mechanism overcomes the control and synchronization problems that are inherent in the relative debugging technique. Finally, a debug back end incorporates data-parallel language support.

Client–server architecture

Programs controlled by a relative debugger can be any combination of sequential and parallel codes. Sequential programs might be specific to a small range of machines, such as particular vector architectures. Many massively parallel processing systems use networks of independent nodes and rely on the implementations of parallel programming models—such as a Message Passing Interface and Parallel Virtual Machine—to manage process creation.

A number of debuggers and debugging environments have been developed to support parallel and distributed archi-

tectures, many of which employ a client–server mechanism.[4–7] Some, such as Detop,[8] support both task and data parallel codes. Guard99 also employs a client–server model to ensure that the processes being debugged can be distributed onto multiple platforms and controlled independently. Figure 1 shows the client–server architecture Guard99 uses.

Figure 2 shows details of the components that make up the debug client. The client user interface accepts and processes user commands in either of two modes: immediate or deferred. In immediate mode, the user interface parses and decodes commands and executes them immediately, displaying any results to the user. In deferred mode, the user interface accepts commands that are in turn passed to the dataflow compiler for translation into a graph. The dataflow engine can then execute this graph at a later date.

The debug client Application Programming Interface layer manages the debug requests from the client, regardless of whether they are a result of immediately executed commands or are generated by the dataflow engine. It also provides a consistent interface to debugging actions that can be performed on target programs. This layer is responsible for translating these actions into the appropriate network requests using the network API layer and for receiving and processing responses from the debug server.

The architecture-independent format API provides an interface for managing and manipulating data in an AIF. All data transmitted between client and server is first converted to this format using routines that the AIF API supports.

The debug servers are responsible for managing the processes that are being debugged. Each server manages exactly one process—so, for example, if a parallel program starts 10 processes, then 10 debug servers also need to be created. Figure 2 also shows the components that constitute a debug server. The server receives requests from the client through the network API layer. These requests are then passed to the debug server API layer, which converts the request into a form suitable for the debug back end. The back end controls the debugger's low-level architecture-specific functions. Maintaining this distinction between the server API and the back end allows for a clean separation between the architecture-independent and architecture-dependent parts of the server. This ensures that additional architecture support can be easily added to the debugger.

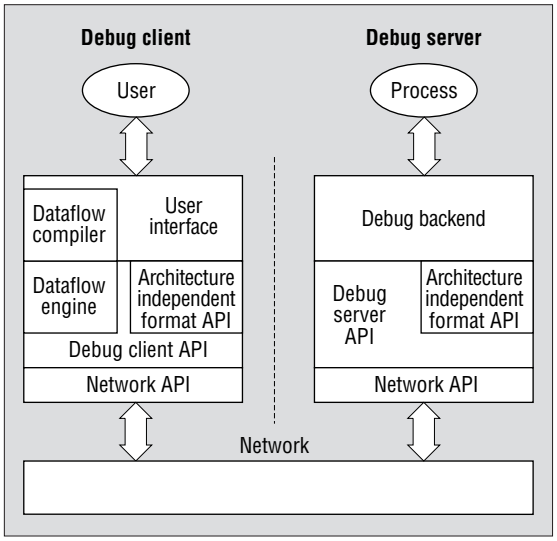


Figure 2. The debugger architecture.

Architecture-independent data format

In addition to debugging programs on physically separate hosts, a relative debugger must also support programs running on heterogeneous architectures. Data from remote systems must be transferred to the client as a result of formatting or display commands, or for arithmetic or comparison operations generated by the dataflow engine when executing assertions. The remote systems (and the client) might each employ different architectural features such as word length and byte ordering.

In the client–server model we present, the client requests data from a remote process by sending a request to the debug server controlling that process. The server first converts the data into an architecture-independent (data) format on the remote system and the formatted data is transferred to the client. The client then uses AIF library routines for all manipulation, arithmetic, and comparison operations performed on this data.

Standard networking protocols such as XDR have addressed the problem of sending architecture-specific data over a network. Much work has also been done on the development of architecture-independent file formats, with the National Center for Supercomputing Applications' hierarchical data format now accepted as the de facto standard. However, none of these approaches address the issues of performing in-memory operations on data from architecturally different systems. As a result, we have provided a well-defined API and have developed library routines that implement all the necessary arithmetic and conversion operations.

AIF achieves architecture independence through format standardization and by employing data tags. Byte size is standardized to eight bits, and integer byte ordering is big-endian (for a given multibyte numeric representation, the most significant byte has the lowest address). Characters are single byte and are expressed in terms of the ASCII collating sequence, and floating-point numbers use the big-endian IEEE 754-1985 format. Simple data types using other architecture-specific data representations are translated into the standard formats when converted to AIF.

When data is converted to the standard format, it is tagged with a format descriptor string that describes the data's size and layout. Table 1 shows the tags that are currently available. Tags for simple data types provide AIF library routines with information such as the size of the data and whether or not the data is signed. This supports different integer sizes and single, double, and extended floating-point formats and ensures that the AIF library routines can perform calculations with no loss of precision. Complex data types have tags that describe the data's size and memory layout and contain nested tag types. Figure 3 shows an example of how a C structure is converted to AIF.

The current implementation provides descriptors for C, Fortran, and ZPL. Support for another language can be added by defining new descriptors for

Table 1. AIF format descriptor tags.

TAG	TYPE	DETAILS
`c`*s*	Character	*s* is `s` (signed) or `u` (unsigned)
`i`*sl*	Integer	*s* is `s` (signed) or `u` (unsigned), *l* is size in bytes
`f`*l*	Floating point	*l* is size in bytes
`^`*lt*	Address	Type *t*, *l* is size of the address in bytes
`[`*t1*`]`*t2*	Array	Type *t2* with index type *t1*, which must be range
{*l* = f_1@o_1#l_1:t_1, ..., f_n@o_n#l_n:t_n}	Structure or union	*l* is size in bytes, f_i is the name of the field, o_i is the offset in bits from structure's origin, l_i is the size in bits, and t_i is type of the field
<e_1=v_1, ..., e_n=v_n>	Enumeration	e_i is the name of each value v_i
`r`v_{min}..v_{max}*t*	Range	Based on integral type *t* with v_{min} and v_{max} as limits
`v`*l*	Void	*l* is size in bytes
`R`*rt*	ZPL region	Rank *r*, whose limits are based on integral type *t*

each of the languages data types that the current descriptors don't cover and by providing routines to perform appropriate operations on these data types. For example, including support for Data Parallel C would require adding a shape descriptor and routines that implement shape semantics to satisfy the AIF API's requirements.

DATAFLOW ENGINE

In general, the processes being controlled are not synchronized. Thus, they might reach breakpoints and send data to the debugger at arbitrary times in relation to each other. Because of this, relative debugging uses a dataflow execution mechanism for processing user-defined assertions. We previously examined the reasons for choosing such a mechanism over more conventional execution mechanisms.[9] Here, we only summarize the concept, which provides an operational semantics for relative debugging constructs in terms of dataflow graphs. In the debugger, the deferred mode command interpreter collects assertions and other control statements and passes them to the dataflow compiler. The compiler translates the assertions into a dataflow graph, which is stored internally. Figure 4 shows a simplified dataflow graph that results from compiling an example assertion. When the user issues an immediate mode command to start the graph, it is passed to the dataflow engine for execution.

Nodes in the dataflow graph *fire* when tokens are present on all their inputs. The result of a node firing is to generate another token. The graph edges specify connections between nodes, which determine where to send the resulting token.

In this example, executing the graph begins by setting breakpoints at line 4,300 in `$proc1` and 4,400 in `$proc2` and by sending the appropriate tokens to a SETBP node. The graph starts executing the programs using the CONT nodes and then waits until the breakpoints are reached using the WAIT nodes. Once a program is stopped at the appropriate breakpoint, data is extracted using the READ node and is sent to the COMP node for comparison. At the same time, sending a token back to the CONT node restarts program execution. The programs are free to resume execution as soon as they restart; however, if they contain a loop structure, they might hit the breakpoints again. Because the dataflow engine will continue to check for programs stopped at a breakpoint, the end result will be a steady stream of tokens reaching the COMP note's inputs. As soon as data becomes available on both inputs of the COMP node, a comparison is performed. If differences exist, the results are sent to the DISP node to be displayed to the user.

This approach's main advantage is that it implements the semantics of assertion statements naturally. Two data structures can only be compared when the data is available from both processes, which in turn depends on when arbitrary breakpoints have been reached. Because a compare node will only fire when tokens are present on both its inputs, the dataflow engine enforces the matching rules automatically.

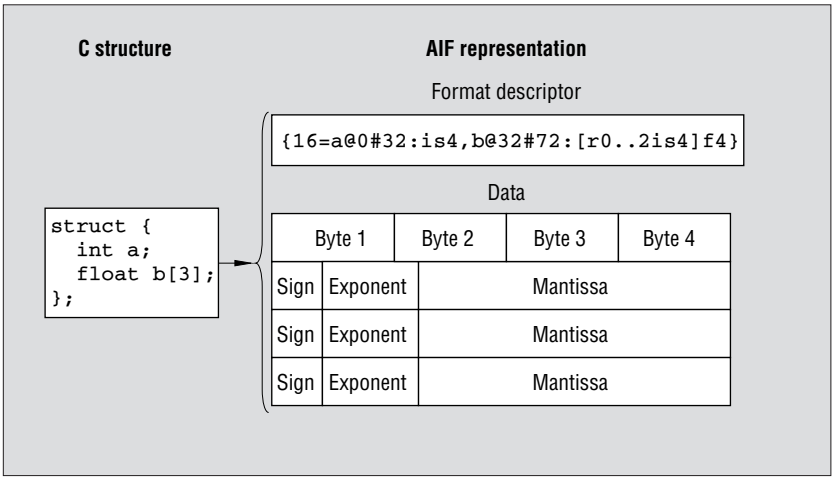


Figure 3. Architecture-independent format conversion example.

DATA-PARALLEL LANGUAGE SUPPORT

ZPL, like other data-parallel languages, relies on the language runtime system to manage the distribution of parallel arrays in a manner that is normally hidden from the user. The user need not be concerned about how blocks of data will be decomposed and mapped to processes (although some languages, such as HPF and Fortran D, provide mechanisms to specify this). Similarly, the user should interact with the debugger in terms of the data structures themselves, without concern for their decomposition and distribution.

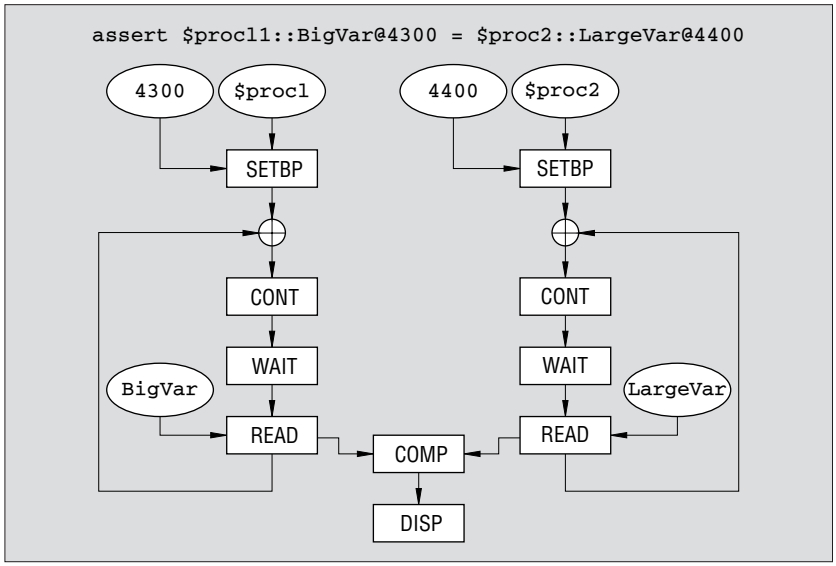


Figure 4. An assertion compiled to a dataflow graph.

Data-parallel language runtime systems generally maintain a per-process data structure that contains a description of each parallel array and provides information such as the array's rank, each dimension's size, and information on the array's distribution across the processes. To access a block of data from an individual process, the debugger must first access this description information to determine the location and bounds of the block that resides in the process. In ZPL, this distribution information is stored in a runtime structure called an *ensemble*, one of which is maintained for each parallel array in each process. An ensemble consists of the following (simplified) structure:

```
struct ensemble {
   int      blocksize [MAXRANK];
   int      offset [MAXRANK];
   int      stride [MAXRANK];
   void *   data;
   int      numdims;
   region * regptr;
   unsigned long   size;
   char *   basetype;
};
```

To access the data specific to a particular process, the debug server for that process must first extract the upper and lower bounds for each index of the array from *regptr*—a pointer to the array's region information. The location of a particular element of the array is then computed from the offset, stride, and blocksize information. For some element in the array whose indices are $(a_0, a_1, \ldots, a_n)$, where n is the number of dimensions specified by `numdims`, the location of the data is given by

$$\text{data} + \sum_{i=0}^{\text{numdins}-1} \times \left(\frac{(a_i - \text{offset}[i])}{\text{stride}[i]} \times \text{blocksize}[i] \right)$$

The debug server interprets this information and uses it to obtain a copy of the data that is specific to the particular process.

To let data-parallel arrays be used in assertions in a transparent manner, Guard99 uses the decomposition information along with the per-process location information to reassemble a complete structure. Because Guard99 knows the language type of the program being debugged, it can apply the language decomposition rules to request the components of the array from each process via the debug servers. It then reassembles these components into a complete array, and assertions can be used to compare the data in this array with that obtained from a reference program.

Adding data-parallel language support to the debugger then becomes a three-step process. First, the syntax and semantics of the language must be defined in the debugger parser. However, because the debugger only allows immediate evaluation of expressions, the full language syntax does not need to be defined. Second, the parallel data distribution information must be made accessible to the debugger back end. Finally, AIF tags might need to be added to support any new data types the language introduces.

For example, to add support for the data-parallel language C* to Guard99, new language support would need to be included in the parser.[10] C* provides additional syntax to allow scalar access of parallel arrays, adds a number of new operators such as minimum and maximum value reduction (`<?=` and `>?=`), and introduces a new type syntax for shape declarations. Next, the runtime representation of parallel variables would need to be added to allow the debugger back end to access the parallel array information. In C*, as in ZPL, a single structure stores the parallel array distribution information. Finally, as for ZPL, a new AIF type would need to be added so that parallel array shape information would be accessible to the debugger client.

Debug back end

The development of Guard99 has been primarily concerned with the technique of relative debugging. We avoided the overheads of developing a machine-specific debug engine by providing a well-defined interface between the debug server and the low-level debug functions. This lets us use a pre-existing debugger, such as GDB, for this purpose.[11]

In addition to providing the low-level debugging functions, using a back end debugger also lets us isolate language-specific details in one place. Accordingly, the debug client and server can be designed to be language independent, relying on the back end to interpret the syntax and semantics of individual languages. Supporting a new language, in this case ZPL, then becomes a process of modifying GDB to interpret the language's specific syntactic and semantic details. This process also has the resulting benefit of enhancing GDB as a standalone debugger.

Simple: A case study

Our case study illustrates the power of relative debugging when applied to two widely different programming languages. It examines the problem of debugging a sequential C code that has been ported to the data-parallel language ZPL. We

Table 2. Scalar error value.

Iteration	C	ZPL (2 x 2)
1	0.984958283	0.984946715
2	0.985004506	0.984971498
3	0.985086136	0.985033153
4	0.985224992	0.985147640

simple.z

```
         program simple;
         ...
         region
            R = [1..DL, 1..DK];
            WEST = [1..DL, 1..DK];
         ...
         direction
            east = [0,+1];
         ...
         var
            X: [R] double;
            Heat: [R] double;
            En_error: [R] double;
            Theta: [R] double;
            Delta_t: [R] double;
            Sc_error: double;
         ...
         procedure InitPositionVelocity()

         begin
154         R := (Index2-1) * deltaR / (maxX - minX) + Rmin;
155         W := (((maxY-Index1) * PI) / (2 * (maxY - minY)))
                 + angleOffset;
156
157         X.r := R * cos(W);
158         X.z := R * sin(W);

         end;
         ...
         procedure simple()
         ...
         [R]
         begin
            /* Initialisation Routines */
            ...
            for count := 0 to N-1 do
               ...
               /* Delta Phase */
               ...
               /* Hydro Phase */
               ...
               /* Heat Phase */
               ...
474            for i := DK-1 downto 0 do
475               [,i] Theta := Alpha * Theta@east + Beta;
476            end;
               ...
479            [WEST] Heat := (Theta - Theta@east) * R_ * Delta_t;
               ...
               /* Energy Phase */
               ...
538            Heat := Heat * Delta_t;
539
540            En_error := Int_en + Kin_en - Work + Heat;
               ...
544            Sc_error := +<< En_error;
               ...
            end;
         end;
```

Figure 5. ZPL code structure.

demonstrate that the architecture proposed earlier is sufficiently powerful to handle the differences in the two languages and their underlying platforms. The case study also highlights the efficiency of the debugging technique for locating errors across multiple program versions.

The program is the Simple hydrodynamics code.[12] To demonstrate our debugging methodology, we chose a problem size of 128×128 elements using four iterations. The "output" of the Simple code is a scalar error value that is calculated from the values used in the hydrodynamics computation. For the C and ZPL comparison, we chose to run the ZPL code using four processes in a 2×2 mesh. The initial run of the codes produced scalar error values that differ at around the fourth decimal place. Table 2 shows the values that were produced for each of the four iterations.

Both the C and ZPL codes employ double-precision variables for all computations, so we would expect the scalar error value to be equivalent to within the precision available, or about 15 decimal places. The precision of the floating-point representation also lets us set a lower bound for the error tolerance used in defining assertions. In this case study, both codes use double precision, so the lower bound will be 10^{-15}. In situations where different precision is used in each code, the lower bound will need to be adjusted to the larger of the two values.

Our initial hypothesis was that the C and the new ZPL codes worked correctly, even though first examination showed that the codes produce slightly different results. To account for the discrepancy, we assumed that different numeric evaluation techniques in the language runtime systems or minor numerical errors were the likely cause. We adopted the three-phase approach when debugging the codes, because it was not obvious at the outset which of these factors contributed to the differences.

The first step compares a single process ZPL code with the same code on multiple processes to ensure that the ZPL runtime system is not introducing any differences into the results. The second step compares the parallel ZPL code to the C code so that errors in the ZPL version of Simple can be identified and corrected. The last step compares the serial ZPL code to the C code as a final check to verify that all errors have actually been corrected.

Relative debugging relies on the abil-

ity to compare a suspect program with a reference code. As a result, a debugging methodology using an iterative refinement process can help narrow the region (or regions) containing potentially incorrect code until the error is located. This process has been applied very successfully in the past in various case studies.[13–15]

Code description

The Simple code models the hydrodynamics of a pressurized fluid inside a spherical shell. The simulation computes values that describe the fluid's physics at many points inside the shell over a number of time steps. Each iteration results in the computation of new values for various physical quantities such as velocity, density, energy, viscosity, pressure, and temperature. Although Simple is modeling the inside of a spherical shell, the problem's symmetry reduces the computation to one quarter of an annular region. This region can then be transformed into Cartesian coordinates so that each physical quantity can be stored in a 2D array.

Figure 5 shows the structure of the ZPL version of the Simple code. The ZPL code combines all the hydrodynamics calculations in a single loop of N iterations, with the scalar error computation performed by lines 540 and 544. The first computes the `En_error` array, and the second performs a reduction across all elements of the `En_error` array to produce the scalar error `Sc_error`.

ZPL introduces parallelism into the Simple code by defining the variables for each physical quantity over a region. The ZPL runtime system can then automatically partition this region into blocks that the individual processes can manage. The algorithm has also been designed so that computations in each phase share the same data dependencies, thus minimizing the overall communications overhead. More details on the ZPL implementation of Simple appear elsewhere.[1]

The C code is structured slightly differently from the ZPL in that each phase of the hydrodynamics calculation is located in a separate module. The main body of the C code (see Figure 6) is contained in the module `simple.c` and consists of an initialization phase, `load()`, followed by N iterations over delta, hydro, heat, and energy phases. The load phase routine `load()` is located in `load.c`, `delta()` in `delta.c`, `heat()` in `heat.c`, and so on.

For the C code, calculation of the scalar error value is performed in `energy()` (see Figure 7). The scalar error value is computed as the sum of all elements in the energy error array `en_error` (line 92), which is in turn derived from the values of the energy phase arrays `int_en, kin_en`, and `work`, and the boundary heat flow array `heat` (line 86). Values for these arrays are computed in the corresponding phase routines.

Serial and parallel ZPL comparison

The first step in the debugging process was to compare the ZPL code in a single-process configuration with that in a multiprocess configuration—in this case, four processes in a 2×2 mesh. The results from these runs showed that different process topologies produced slight variations in the scalar error value, with a magnitude of slightly greater than 10^{-15}. Because differences of this magnitude are still significant for double-precision floating-point numbers, these appeared to be errors introduced by the parallel runtime system.

Guard99 was then used to determine the cause of these differences by defining assertions over the phase variables `Int_en`, `Kin_en`, `Work`, and `Heat` and the error value `En_error`. However, no differences were visible in these variables. This meant that the source of the variations must be the final reduction operation at the end of each iteration. As the order of the floating-point operations is the only factor affected by topology changes, it is likely that the nonassociative nature of these operations caused the variations. This result is important, because it shows

simple.c

```
double x[DL+2][DK+2];
double heat[DL][DK];
double en_error[DL][DK];
double theta[DL][DK];
double delta_t[DL][DK];

main()
{
   int loop = 0;

   ...
   load();
   ...
   do
   {
     delta();
     hydro();
     heat();
     energy();
     ...
   } while (loop++ < N);
   ...
}
```

Figure 6. C code structure.

energy.c

```
   extern double heat[DL][DK];
   extern double delta_t[DL][DK];
   double int_en[DL][DK];
   double kin_en[DL][DK];
   double work[DL][DK];
   ...
   energy()
   {
      int i, j;
      double local_error_sum = 0.0;
      ...
81    for (i=0; i<DL; i++) {
82      for (j=0; j<DK; j++) {
83        heat[i][j] *= delta_t[i][j];
86        en_error[i][j] = int_en[i][j] +
            kin_en[i][j] + work[i][j] +
            heat[i][j];
87      }
88    }
89
90    for (i=0; i<DL; i++) {
91      for (j=0; j<DK; j++) {
92        local_error_sum +=
            en_error[i][j];
93      }
94    }
      ...
   }
```

Figure 7. C scalar error calculation.

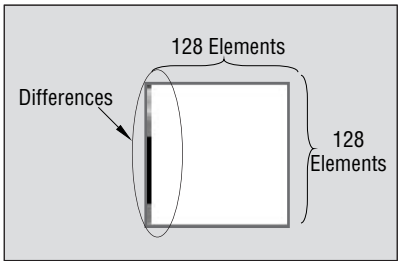


Figure 8. Differences between `$c::heat` and `$zpl::Heat`.

that nondeterministic behavior can affect the computation's results and effectively set a lower bound to the final error value's accuracy.

ZPL and C comparison

Having isolated the variations the parallel nondeterminism introduced, we could identify the cause of the differences between the ZPL and C codes. The next step of the debugging process involved using an iterative refinement process to identify and correct four errors in the code.

Error 1: Extra term in an expression

Debugging the ZPL and C Simple codes starts by defining assertions for the four phase variables used in the scalar error calculation. Because the magnitude of the error was around 10^{-4}, we also set the initial error tolerance to $10^{-5} \le \varepsilon \le 10^{-1}$:

```
set error 1.0e-5 1.0e-1
assert
  $zpl::Int_en@"simple.z":540 =
  $c::int_en@"energy.c":81
assert
   $zpl::Kin_en@"simple.z":540 =
   $c::kin_en@"energy.c":81
assert
   $zpl::Work@"simple.z":540 =
   $c::work@"energy.c":81
assert
   $zpl::Heat@"simple.z":540 =
   $c::heat@"energy.c":81
```

The results from these assertions indicate differences in the `$c::heat` and `$zpl::Heat` arrays, which store the results of the heat phase computation. Figure 8 shows a visualization of these differences. Because the variables are 2D arrays, it is convenient to visualize the differences as a 2D bitmap. These bitmaps are generated from difference information by assigning a color to represent the difference's magnitude, ranging from blue for the smallest difference, through green, yellow, and red, to black for the largest. White indicates no difference.

The differences appear along the left (or western) edge of the comparison. The western edge of the array is accessed in ZPL using the syntax `[WEST] Heat := ...`, where `WEST` has been defined as the appropriate region. A search of the ZPL code results in only one example of such syntax at line 479 in `simple.c`. The corresponding C code can be seen at lines 163–165 of `heat.c` in Figure 9.

Careful examination of both codes indicates that the term `delta_t` (shown in bold in the ZPL code) was erroneously included in the computation of `Heat` in the ZPL code. Prior to using Guard99, this error was not detected, even though it is obvious post fact. In this case, relative debugging let us identify a faulty statement in the ZPL code fragment, even though the syntax and implementation details of the languages completely differ.

heat.c

```
     extern double heat[DL][DK];
     extern double theta[DL][DK];
     double temp_theta[DL][DK];
     ...
     heat()
     {
       ...
143    for (i=DK-2; i>=0; i--) {
144     for (j=0; j<DL; j++) {
145      temp_theta[j][i] = theta[j][i+1];
146     }

149     for (j=0; j<DL; j++) {
150      theta[j][i] = alpha[j][i] * temp_theta[j][i]
             + beta[j][i];
151     }
152   }
      ...
163   for (i=0; i<DL; i++) {
164     heat[i][0] = (theta[i][0] - temp_theta[i][0])
             * r[i][0];
165   }
      ...
     }
```

Figure 9. C heat phase code.

Error 2: Incorrectly specified constant

After correcting the first error, there are still differences visible in the output, though the magnitude has now been reduced to around 10^{-7}. Setting the error tolerance to $10^{-10} \le \varepsilon \le 10^{-1}$ and rerunning the original assertions now indicates differences between `the $c::int_en` and `$zpl::Int_en` arrays, which store the internal energy values computed in the energy phase. Figure 10 shows these differences.

By defining additional assertions, we can observe that differences in many of the variables involved in the computation of the internal energy have similar characteristics to those in Figure 10. With differences occurring in so many variables, no clear path to the error's source is evident. Instead, we chose to examine initialization code for similar patterns of differences, beginning with the code to initialize the position and velocity components used throughout

the program. Figure 11 shows the C version of this code, which corresponds to lines 154–158 of the ZPL code (Figure 5).

Visualizing differences in variables from these codes shows the characteristic pattern in the differences between `$c::w` and `$zpl::W`, and `$c::x` and `$zpl::X`. As both `$c::w` and `$zpl::W` are computed entirely from constants, the problem must relate to these variables. Further examination indicates that the value of `PI` used in the ZPL code is only specified to seven decimal places, while the corresponding value used in the C code, `M_PI`, is specified to 20 decimal places (of which only 15 are significant).

Like the previous error, the difference in `PI` is obvious with hindsight. However, because a symbolic constant is used in the code, a cursory examination would not have revealed the difference. Relative debugging let us identify a characteristic pattern of differences that was visible in a number of variables, suggesting that a common source was responsible. The error was eventually located by tracing this pattern back to the constant declarations, even though the two languages use different syntactic structures for defining constants.

Error 3: Invalid boundary conditions

There are still differences in the output of the programs, although the magnitude has now been further reduced to around 10^{-10}. Setting the error tolerance to $10^{-15} \le \varepsilon \le 10^{-1}$ and rerunning the original assertions shows that the `$c::heat` and `$zpl::Heat` arrays are again the source of errors. A series of assertions must now be applied to narrow down the erroneous region of code in the heat phase computation. The result of these assertions is that the problem appears to be occurring between lines 474 and 479 of the ZPL code and lines 143 and 152 of the heat phase calculation in `heat.c`.

The ZPL code uses the `(i+1)st` column of `Theta` in the computation `Alpha * Theta@east + Beta` and propagates this across columns `(DK-1)` to `0` of `Theta`. The C code uses a temporary array to hold the `(i+1)st` column of `Theta`. However, the outer loop of the C code only ranges from `(DK-2)` to `0`, so the `(DK-1)st` column is not computed, and hence the C code is incorrect. An identical situation is also found in the computation of the north boundary condition.

This error is interesting because the ZPL code is actually correct, while the original C code is incorrect. When developing the ZPL code, the programmer did not need to be concerned with issues such as computing loop bounds, but instead was able to concentrate on the underlying physics of the model. In comparison, the C programmer needed to consider the loop-bound issues, with the extra complexity presumably leading to the coding error. In spite of these significant implementation differences, relative debugging identified the incorrect code.

Error 4: Wrong sign

Even after correcting the third error, the magnitude of the differences in the output of the programs still remains at around 10^{-10}. Further examination of the scalar error computation indicates that while the values of `$c::en_error` and `$zpl::En_error` have very small differences, the calculations of the energy, work, and heat values are now identical. This can only point to a problem with calculating the error value itself. Close examination of the ZPL and C code shows the source of the error, which can be seen at line 540 of the ZPL code and line 86 of the C code in Figure 11.

Using relative debugging, we identify a pattern of differences and quickly pinpoint the location of erroneous code that is only obvious with the benefit of hindsight. Interestingly, the original paper[12] describing the Simple code gives the error computation as

```
Int_en + Kin_en – Work – Heat.
```

This means that both programs are actually incorrect.

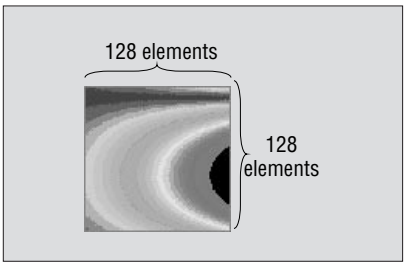


Figure 10. Differences between `$c::int_en` and `$zpl::Int_en`.

SERIAL ZPL AND C COMPARISON

The final step in the debugging process is to verify that the changes made to both the ZPL and C codes resulted in bitwise equivalence of the variables used in the physics computation. Because nondeterminism was introduced by running the ZPL code in parallel, we must compare the serial ZPL and C codes. For this test, we used the same series of assertions defined over the four phase variables and set the error tolerance to 0 (although a tolerance of 10^{-16} would have been equally valid). As predicted, the results showed that each of the variables were now identical.

The debugging exercise outlined in this section took a remarkably short time, considering that the programmer performing the case study was not the author of either version of Simple and was not particularly fluent in ZPL. Although it is dangerous to generalize

load.c

```
101   for (i=0; i<DL+2; i++) {
102     for (j=0; j<DK+2; j++) {
103       r[i][j] = ((PEj-1)*DK + (j-1)) * deltaR /
                  (NUM_K_PROCS * DK - 1) + Rmin;
104       w[i][j] = ((M_PI * ((PEi-1) * DL + (DL-i)))
           / (2 * (NUM_L_PROCS * DL-1))) + ANGLE_OFFSET;
          ...
107       x[i][j].r = r[i][j] * cos(w[i][j]);
108       x[i][j].z = r[i][j] * sin(w[i][j]);
109     }
110   }
```

Figure 11. C position and velocity initialization code.

the results too far, this example adds to the evidence of our other case studies that supports the power of relative debugging—in the case even when the programs run on different computers and are written in different languages.

DESPITE ITS POWER, comparing data by specifying variable and breakpoint information in assertion statements is still a very low-level approach. It relies on the programmer making sensible choices about the location from which data will be extracted. To compare data, it is not only important that the programs are functionally equivalent at this point, but that breakpoints aren't inserted at locations where the control flow is disrupted or data is in an indeterminate state. This issue is particularly important for arbitrary parallel programs incorporating distributed processes or multithreading and requires further investigation. Moreover, relative debugging cannot currently be applied to find timing errors, which are a common cause of failure in task-parallel programs. In fact, the insertion of data-gathering breakpoints, as a relative debugger requires, alters the timing of the programs and might mask or highlight timing problems. It might be possible to combine the assertion constructs used here with data-gathering techniques, which are not as invasive as the current debug server, but this also requires further investigation.

Guard99 has been used extensively to examine the relationships between separately executing programs, but there is some evidence that the relative debugging methodology could also be useful for monitoring information in separate *processes* of a parallel or multithreaded program—for example, to ensure that data is initialized correctly or that message buffers are in a consistent state.

Here, we addressed the use of a relative debugger for data-parallel programs. However, we have also implemented a scheme letting the user describe the distribution of data structures for arbitrary task-parallel programs. The system uses a declarative algebraic approach and lets the program ignore the decomposition when formulating the assertions. It also lets a user describe changes that occur in a data structure and associated code when a program is modified for parallel execution.

Finally, a relative debugger is only one tool that can help find errors. An extremely powerful interactive environment could be built by combining a relative debugger with a call tree browser and dataflow analyzer. Using such a tool would make it easier to trace error propagation through a program, because the user could trace the source of errors in one variable to those that are used to compute the state. In fact, such an approach might facilitate automatic iterative refinement by traversing a program's dataflow. Such an exciting prospect certainly requires further investigation.

Acknowledgments

The Australian Research Council, under the Large Grants Scheme, supported this work. We would also like to acknowledge our colleagues Larry Snyder, Bradford Chamberlain, and Sung-Eun Choi at the University of Washington for the assistance they provided in obtaining the results we reported in this article.

References

1. C. Lin and L. Snyder, "ZPL: An Array Sublanguage," *Languages and Compilers for Parallel Computing, 6th Int'l Workshop Proc.*, Springer-Verlag, Berlin, 1994, pp. 96–114.
2. B. Chamberlain et al., "Factor-Join: A Unique Approach to Compiling Array Languages for Parallel Machines," *Workshop on Languages and Compilers for Parallel Computing, Proc. 9th Int'l Workshop*, Springer-Verlag, Berlin, 1997, pp. 481–500.
3. L. Snyder, *A Programmer's Guide to ZPL*, MIT Press, Cambridge, Mass., 1999.
4. D. Cheng and R. Hood, "A Portable Debugger for Parallel and Distributed Programs," *Proc. Supercomputing '94*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1994, pp. 723–732.
5. P. Kacsuck et al., "A Graphical Development and Debugging Environment for Parallel Programs," *Parallel Computing*, Vol. 22, No. 13, 1997, pp 1747–1770.
6. J. May and F. Berman, "Panorama: A Portable Extensible Parallel Debugger," *SIGPLAN Notices*, Vol. 28, No. 12, San Diego, 1993, pp. 96–106.
7. *TotalView Multiprocess Debugger Users' Guide Version 3.7.7*, Dolphin Interconnect Solutions, Rev. 8, Oct. 1997.
8. T. Bemmerl and R. Wismüller, "On-Line Distributed Debugging on Scaleable Multicomputer Architectures," *High Performance Computing and Networking, Volume II: Networking Tools, Lecture Notes in Computer Science*, Vol. 797, Springer-Verlag, New York, 1994, pp. 394–400.
9. D. Abramson, R. Sosic, and G. Watson, "Implementation Techniques for a Parallel Relative Debugger," *Proc. PACT '96*, IEEE Computer Soc. Press, Los Alamitos, Calif, 1996.
10. J. Rose and G. Steele Jr., *C*: An Extended C Language for Data Parallel Programming*, Tech. Report PL 87-5, Thinking Machines Corp., Cambridge, Mass., 1987.
11. R. Stallman, *Debugging with GDB—The GNU Source Level Debugger, Edition 4.12*, Free Software Foundation, Boston, 1994.
12. W. Crowley, C. Hendrickson, and T. Rudy, *The SIMPLE Code*, Lawrence Livermore Laboratory, Livermore, Calif., UCID-17715, 1978.
13. D. Abramson and R. Sosic, "A Debugging and Testing Tool for Supporting Software Evolution," *Automated Software Eng.*, Vol. 3, Nos. 3–4, Aug. 1996, pp. 369–390.
14. D. Abramson and R. Sosic, "A Debugging Tool for Software Evolution," *CASE-95, Seventh Int'l Workshop Computer-Aided Software Engineering, Proc. 7th Int'l Workshop Computer-Aided Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1995, pp 206–214.
15. D. Abramson et al., "Relative Debugging: A New Methodology for Debugging Scientific Applications," *Comm. ACM*, Vol. 39, No. 11, Nov. 1996, pp. 69–77.

Greg Watson is a Senior Research Fellow at the Monash University School of Computer Science and Software Engineering. His research interests include programming tools for parallel and distributed computers, and parallel languages and runtime systems. He is also interested in operating system design, nano technology, and AI. He is currently Vice-President of the Internet Society of Australia and Chair of the Australian .au Domain Administration (auDA). Contact him at greg@csse.monash.edu.au

David Abramson is head of the School of Computer Science and Software Engineering at Monash University, Australia and is project leader in the Cooperative Research Centre for Distributed Systems Nimrod Project and the Australian Research Council funded Guard project. His current interests are in high performance computer systems design and software engineering tools for programming parallel and distributed supercomputers. Contact him at davida@csse.monash.edu.au.

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Future Generation Computer Systems 19 (2003) 665–678

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/future

Debugging scientific applications in the .NET Framework

David Abramson*, Greg Watson[1]

School of Computer Science and Software Engineering, Monash University, Clayton, Vic. 3800, Australia

Abstract

The Microsoft .NET Framework represents a major advance over previous runtime environments available for Windows platforms and offers a number of architectural features that would be of value in scientific programs. However there are such major differences between .NET and legacy environments under both Windows and UNIX, that the effort of migrating software is substantial. Accordingly, software migration is unlikely to occur unless tools are developed for supporting this process. In this paper we discuss a 'relative debugger' called Guard which provides powerful support for debugging programs as they are ported from one environment or platform to another. We describe a prototype implementation developed for Microsoft's Visual Studio.NET—a rich interactive environment that supports code development for the .NET Framework. The paper discusses the overall architecture of Guard under VS.NET and highlights some of the technical challenges that were encountered during its development. A simple case study is provided that demonstrates the effectiveness of relative debugging in locating subtle errors that occur when even a minor upgrade is attempted from one version of a language to another. For this example, we illustrate the use of relative debugging using a Visual Basic program that was ported from Visual Basic 6.0 to Visual Basic.NET.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Microsoft .NET Framework; Common Language Specification; Relative debugger

1. Introduction

The .NET Framework is a major initiative by Microsoft that provides a uniform multi-lingual platform for software development [17]. It is based on a Common Language Specification (CLS) that supports a wide range of programming languages and runtime environments. In addition, it integrates web services in a way that facilitates the development of flexible and powerful distributed applications. Clearly this has applicability in the commercial domain of e-commerce and P2P networks which rely primarily on distributed applications.

An analysis of the features available in .NET suggests that the new architecture is as equally applicable to scientific computing as to commercial applications. In particular .NET provides efficient implementations of a wide range of programming languages, including FORTRAN [12], because it makes use of just-in-time compilation strategies. Further, the Visual Studio development environment is a rich platform for performing software engineering as it supports integrated code development, testing and debugging from the one tool.

Some of the more advanced features of .NET, such as Web Services, could also have interesting application in scientific code. For example it would be

* Corresponding author. Tel.: +61-03-9905-1183; fax: +61-03-9905-5146; mobile: +61-0417-375-635.
E-mail addresses: davida@csse.monash.edu.au (D. Abramson), gwatson@lanl.gov (G. Watson).
URLs: http://www.csse.monash.edu.au/~davida, http://www.acl.lanl.gov/cluster/people/gwatson
[1] Present address: MS B287, Advanced Computing Lab, Los Alamos National Laboratory, Los Alamos, NM 87545, USA.

0167-739X/02/$ – see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0167-739X(02)00176-0

possible to source libraries dynamically from the Web in the same way that systems like NetSolve [7] and NEOS [9] provide scientific services remotely. This functionality could potentially offer dramatic productivity gains for scientists and engineers, because they can focus on the task at hand without the need to develop all of the support libraries.

Unfortunately the differences between .NET and other legacy software systems, such as WIN32 and even UNIX, are substantial and as a result there is a significant impediment to porting codes from one environment to another. Not only are the environments different functionally, but the libraries and machine architectures may differ as well. It is well established that different implementations of a programming language and its libraries can cause the same program to behave erroneously. Because of this the task of moving code from one environment to another can be error prone and expensive. Many of these applications may also be used in mission critical situations like nuclear safety, aircraft design or medicine, so the cost of incorrect software can potentially be enormous. Unless software tools are developed that specifically help users in migrating software to the .NET Framework, it is likely that most scientists will continue to use legacy platforms for their software development.

Traditional debuggers are not particularly helpful at finding errors introduced during the porting process even when they have been designed with scientific or distributed computing in mind [6,8,15,16,19,25]. This is because they generally require the programmer to have a good understanding of the way the program works and have a mental model of the contents of the various data structures during execution. In this paper we describe a debugging tool called Guard, which specifically supports the process of porting codes from one language, operating system or platform to another. Guard has been available under UNIX for some time now, and we have proven its applicability for assisting the porting of programs many times. We have recently implemented a version of Guard that is integrated into the Microsoft Visual Studio.NET development environment. Not only can the system be used to support porting from WIN32 to .NET, but we have even demonstrated the ability to support cross-platform debugging between a UNIX platform and a Windows platform. This has shown that the tool is not only useful for supporting software development on the one platform, but can also support the porting of codes between Windows and UNIX.

The paper begins with a discussion of the Guard debugger, followed by a description of the .NET Framework. We then describe the architecture of Guard as implemented under Visual Studio.NET, and illustrate its effectiveness in locating programming errors in this environment.

2. Guard—a relative debugger

Relative debugging was first proposed by Abramson and Sosic in 1994. It is a powerful paradigm that enables a programmer to locate errors in programs by observing the divergence of key data structures as the programs are executing [1–5,20,24]. The technique of relative debugging allows the programmer to make comparisons of a *suspect* program against a *reference* code. It is particularly valuable when a program is ported to, or rewritten for, another language or computer platform. Relative debugging is effective because the user can concentrate on *where* two related codes are producing different results, rather than being concerned with the actual values in the data structures. Various case studies reporting the results of using relative debugging have been published [1–3,14,24], and these have demonstrated the efficiency and effectiveness of the technique. The concept of relative debugging is both language and machine independent. It allows a user to compare data structures without concern for the implementation, and thus attention can be focussed on the cause of the errors rather than implementation details.

To the user, a relative debugger appears as a traditional debugger, but also provides additional commands that allow data from different processes to be compared. The debugger is able to control more than one process at a time so that, once the processes are halted at breakpoints, data comparison can be performed. There are a number of methods of comparing data but the most powerful of these is facilitated by a user-supplied *declarative assertion*. Such an assertion consists of a combination of data structure names, process identifiers and breakpoint locations. Assertions are commands that are processed by the debugger before program execution commences and used to build an internal graph [5] which describes when the two

programs must pause, and which data structures are to be compared. In the following example:

```
assert $reference::Var1@1000
  = $suspect::Var2@2000
```

the `assert` statement compares data from `Var1` in `$reference` at line 1000 with `Var2` in `$suspect` at line 2000. A user can formulate as many assertions as necessary and can refine them after the programs have begun execution. This makes it possible to locate an error by placing new assertions iteratively until the suspect region of code is small enough to inspect manually. This process is incredibly efficient. Even if the programs contain millions of lines of code the debugging process refines the suspect region in a binary fashion so it only takes a small number of iterations to reduce the region to a few lines of code.

Our implementation of relative debugging is embodied in a tool called Guard. We have produced implementations of Guard for many varieties of UNIX, in particular Linux, Solaris and AIX. A parallel variant is available for debugging applications on shared memory machines, distributed memory machines and clusters. Currently this is supported with UNIX System V shared memory primitives, the MPICH library, as well as the experimental data parallel language ZPL [24].

The UNIX versions of Guard are controlled by a command line interface that is similar in appearance to debuggers like GDB [21]. In this environment an assert statement such as the one above is typed into the debug interpreter and must include the actual line numbers in the source as well as the correct spelling of the variables. As discussed later in the paper Guard is now integrated into the Microsoft Visual Studio environment and so is able to use the interactive nature of the user interface to make the process of defining assertions easier.

3. Success stories

Over the last few years we have used Guard to debug a number of scientific codes that have been migrated from one platform to another or from one language to another (or both). In one case study we used Guard to isolate some discrepancies that occurred when a global climate model was ported from a vector architecture to a parallel machine [2]. This study illustrated that it is possible to locate subtle errors that are introduced when programs are parallelised. In this case both models were written in the same language, but the target architecture was so different that many changes were required in order to produce an efficient solution. Specifically, the mathematical formulation needed to be altered to reduce the amount of message passing in the parallel implementation, and other changes such as the order of the indexes on key array data structures needed to be made to account for an RISC architecture as opposed to a vector one.

In another case study we isolated errors that occurred when a photo-chemical pollution model was ported from one sequential workstation to another [1]. In this case the code was identical but the two machines produced different answers. The errors were finally attributed to the different behaviour of a key library function, which returned slightly divergent results on the two platforms.

In a more recent case study we isolated errors that occurred when a program was rewritten from C into another language, ZPL, for execution on a parallel platform [24]. This case study was interesting because even though the two codes were producing slightly different answers, the divergence was attributed to different floating point precision. However by using Guard it was possible to show that there were actually four independent coding errors—three in the new ZPL program, and surprisingly, one in the original C code.

All of these case studies have highlighted the power of relative debugging in the process of developing scientific codes. We believe that many of the same issues will arise when migrating scientific software to the new .NET Framework and that Guard will be able to play an important role in assisting this process.

4. The .NET Framework

The Microsoft .NET Framework represents a significant change to the underlying platform on which Windows applications run [17]. The .NET Framework defines a runtime environment that is common across all languages. This means that it is possible to write applications in a range of languages, from experimental research ones to standard production ones, with the

expectation that similar levels of performance and efficiency will be achieved. An individual program can also be composed of modules that are written in different languages, but that interoperate seamlessly. All compilers that target the .NET environment generate code in an Intermediate Language (IL) that conforms to a CLS. The IL is in turn compiled into native code using a just-in-time compilation strategy. These features mean that the .NET Framework should provide an efficient platform for developing computational models.

The Web Services features of .NET also offer significant scope for scientific applications. At present most computational models are built as single monolithic codes that call library modules using local procedure calls. More recent developments such as the NetSolve and NEOS application servers have provided an exception to this strategy. These services provide complex functions such as matrix algebra and optimisation algorithms using calls to external servers. When an application uses NetSolve, it calls a local 'stub' module that communicates with the NetSolve server to perform some computation. Parameters are sent via messages to the server and results are returned the same way. The advantage of this approach is that application programmers can benefit by using 'state of the art' algorithms on external high-performance computers without the need to run the codes locally. Further, the load balancing features of the systems are able to allocate the work to servers that are most lightly loaded. The major drawback of external services like this is that the application must be able to access to required server and so network connectivity becomes a central point of failure. Also, building new server libraries is not easy and requires the construction of complex web hosted applications. The .NET Framework has simplified the task of building such servers using its Web Services technology. Application of Web Services to science and engineering programs is an area of interest that requires further examination.

Visual Studio.NET (VS.NET) is the preferred code development environment for the .NET Framework. The VS.NET environment represents a substantial change to previous versions of Visual Studio. Older versions of Visual Studio behaved differently depending on the language being supported—Visual Basic used a different set of technologies for building applications than Visual C++. The new VS.NET platform has been substantially re-engineered and as a consequence languages are now supported in a much more consistent manner.

VS.NET also differs from previous versions by exposing many key functions via a set of remote APIs known as 'automation'. This means that it is possible to write a third party package that interacts with VS.NET. For example, an external application can set breakpoints in a program and start the execution without user interaction. A separate Software Development Kit (SDK) called VSIP (Visual Studio Integration Program) makes it possible to embed new functions directly into the environment. This allows a programmer to augment VS.NET with new functionality that is consistent with other functions that are already available and operates seamlessly with the user interface. This feature has allowed us to integrate a version of Guard with Visual Studio as discussed in the next section.

5. Architecture of guard

Fig. 1 shows a simplified schematic view of the architecture of Guard under VS.NET. VS.NET is built around a core 'shell' with functionality being provided by commands that are implemented by a set of 'packages' These packages are conventional COM objects that are activated as a result of user interaction (such as menu selection) within VS.NET, and also when various asynchronous events occur. This component architecture makes it possible to integrate new functionality into the environment by loading additional packages.

Debugging within the VS.NET environment is supported by three main components. The Debugger package provides the traditional user interface commands such as 'Go', 'Step', 'Set Breakpoint', etc. that appear in the user interface. This module communicates with the Session Debug Manager, which in turn provides a multiplexed interface into one or more per-process Debug Engines. The Debug Engines implement low-level debug functions such as starting and stopping a process, setting breakpoints, and providing access to the state of the process. Debug Engines can cause events to occur in response to conditions such as a breakpoint being reached and these are passed back through the Session Debug Manager to registered event handlers.

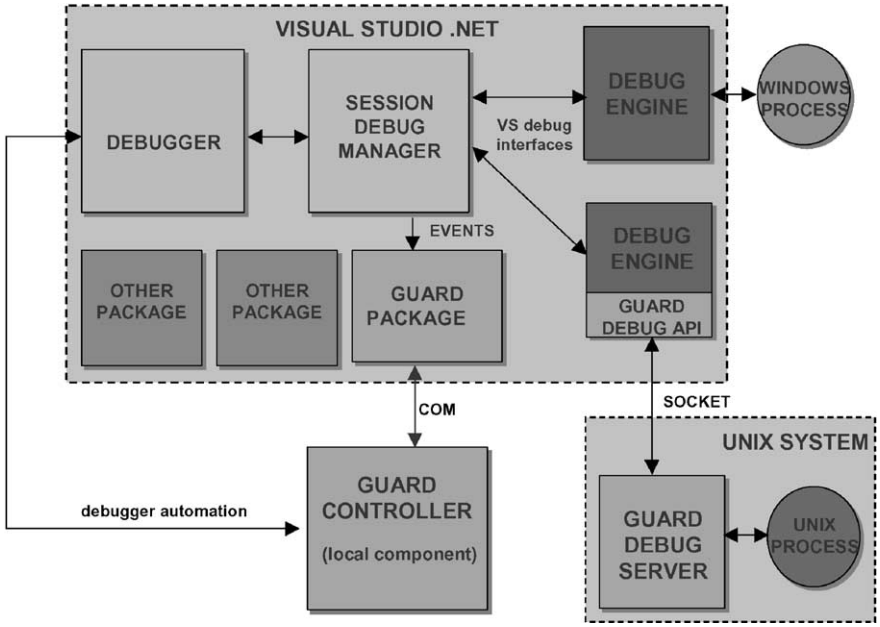


Fig. 1. Guard architecture.

Each Debug Engine is responsible for controlling the execution of a single process. The VS.NET architecture also supports the concept of remote debugging, so a process being debugged may be running on a remote Windows system.

The VS.NET implementation of Guard consists of three main components. A package is loaded into the VS.NET shell that incorporates logic to respond to specific menu selections and handle debugger events. This package executes in the main thread of the shell and therefore has had to be designed to avoid blocking for any extended time period. The main relative debugging logic is built into a local COM component called the Guard Controller. This is a separate process that provides a user interface for managing assertions and a dataflow interpreter that is necessary to implement relative debugging. Because the Guard Controller runs as a separate process it does not affect the response of the main VS.NET thread. The Guard Controller controls the programs being debugged using the VS.NET automation interface. We have also built a Debug Engine that is able to control a process running on an external UNIX platform. This works by communicating with the remote debug server developed for the original UNIX version of Guard using a TCP/IP socket and a custom protocol. The UNIX debug server, based on the GNU GDB debugger, is available for most variants of UNIX, and provides basic debug functions, including process startup to Guard. We have modified GDB to provide support for an Architecture Independent Format (AIF) [24] for data structures, which means it is possible to move data between machines without being concerned about different architectural characteristics, such as word size, endian'ness, etc. AIF also facilitates machine independent comparison of data structures. It is the addition of this Debug Engine that allows us to compare programs executing on Windows and UNIX platforms.

The architecture of Guard is consequently very flexible and allows debugging distributed processes as well as monolithic codes. For example, if an application were decomposed into a number of distinct processes, possibly implemented as Web services, then it would be possible to specify assertions between these individual components and the original sequential code version.

Fig. 2 shows a screen dump of Guard running under VS.NET. When a user wishes to compare two running programs they must first be loaded into a VS.NET 'solution' as separate 'projects'. The solution is then

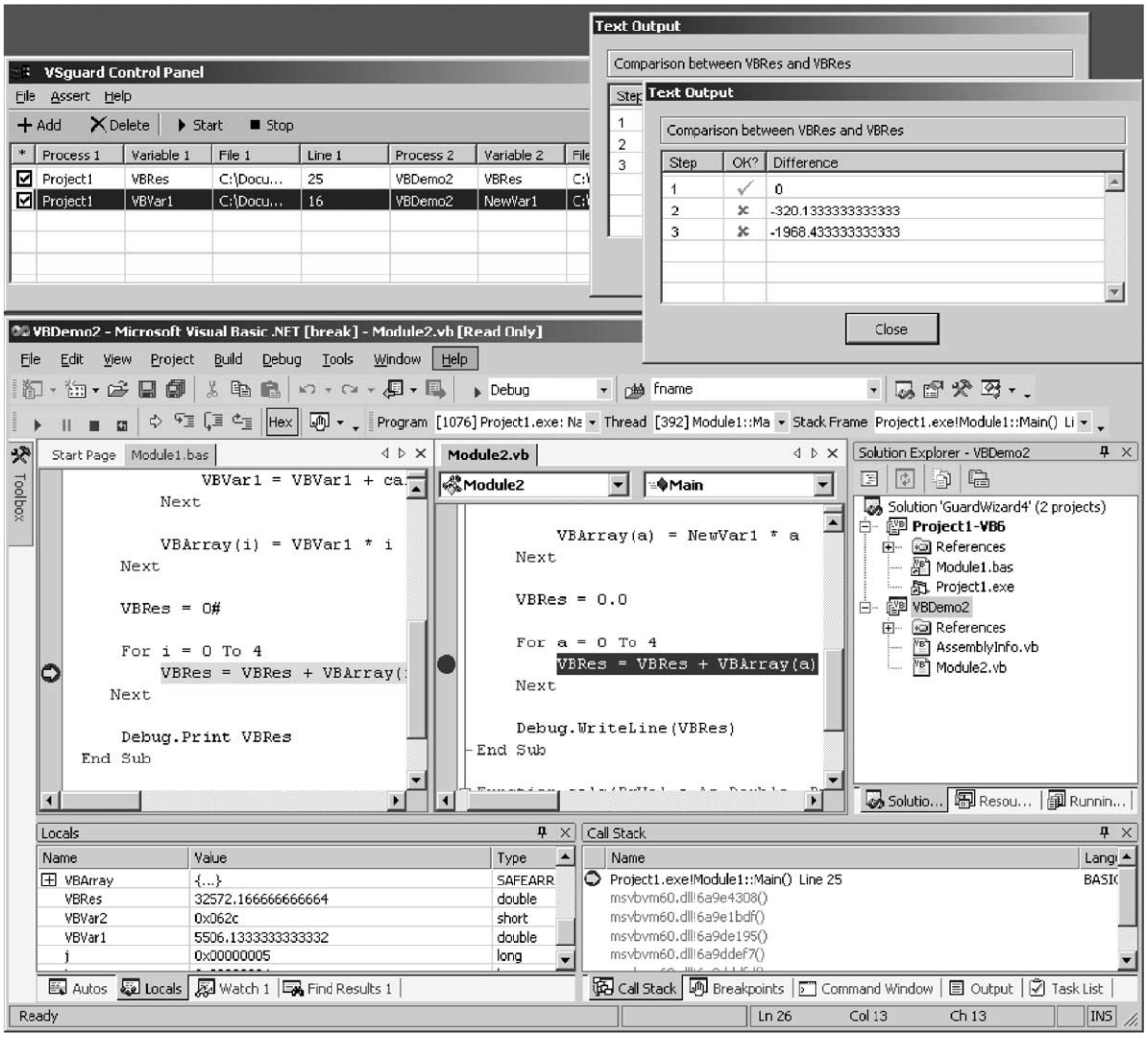


Fig. 2. Guard controller and VS.NET.

configured to start both programs running at the same time under the control of individual Debug Engines. The source windows of each project can then be tiled to allow both to be displayed at once.

A user creates an assertion between the two programs using the Guard Controller, which is started by selecting the ‘VSGuard’ item from the ‘Tools’ menu. The Guard Controller has a separate Control Panel window as shown. An assertion is created in a few simple steps. A new, empty, assertion is created by selecting the ‘Add’ button. Guard displays the dialog box shown in Fig. 3, which allows the user to enter the information necessary to create an assertion. The left-hand side of the assertion can be automatically populated with the variable name, line number, source file and program information by selecting the required variable in the appropriate source window and then using a single right-mouse click. The right-hand side of the assertion can be filled in using the same technique in the other source window. Finally the user is able to specify properties about the assertion such as the error value at which output is generated, when the

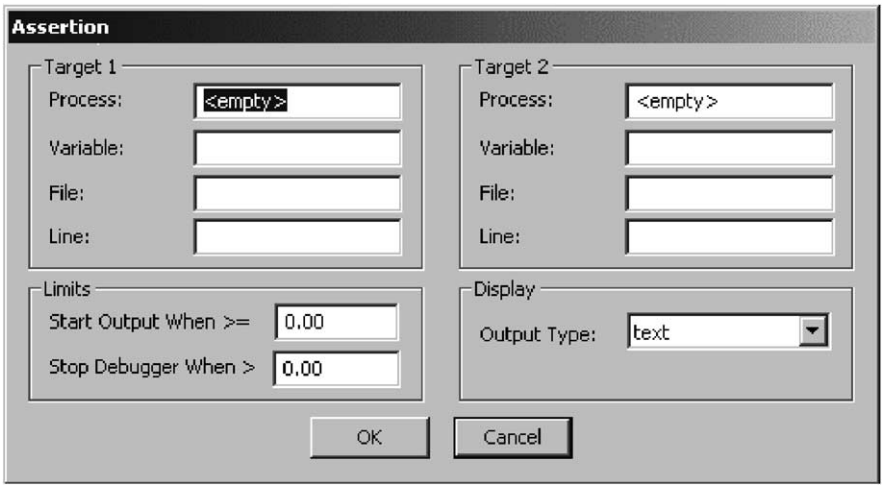


Fig. 3. New assertion dialog.

debugger should be stopped and the type of output to display. The user can create any number of assertions by repeating this process and then launch the programs using the 'Start' button on the Control Panel.

Before commencing execution Guard automatically sets breakpoints at the locations in the source files specified by the assertions. During execution Guard will extract the contents of a variable when its corresponding breakpoint is reached and then perform a comparison once data from each half of the assertion has been obtained. Once the appropriate error threshold has been reached (as specified in the assertion), Guard will either display the results in a separate window or stop the debugger to allow interactive examination of the programs' state. Guard currently supports a number of display types including text, bitmaps and the ability to export data into a visualisation package.

6. Implementation issues

While the VS.NET debugger architecture has been designed with the ability to manage and debug multiple processes, it is not a true multi-process debugger. This is because:

(a) the debugger does not provide control operations, such as start/restart and single step on a per-process basis. Instead these functions only operate on all processes collectively; and
(b) the debugger is constrained to operate in one of two modes where a break event (such as a breakpoint being reached) in one process stops all processes together, or a break event only stops execution of the current process but the debugger is unable to access the other process to obtain state information.

In contrast, the existing Guard architecture assumes that independent control of individual processes is provided by the debugger infrastructure. To address this issue we have modified Guard so that processes are restarted as soon as possible after a breakpoint is reached and data has been extracted. However, because a restart command is issued to all processes Guard must also keep a record of the state of each process so that it can ensure that restart commands are only issued at the appropriate time. The result of this modification is some loss of functionality over the existing UNIX version. In particular it is not possible to stop both programs at a known location when an assertion threshold is exceeded, complicating the manual debugging that might occur after an assertion has triggered.

Our original intention was to integrate both the user interface and a dataflow interpreter into a single multi-threaded package in VS.NET. However

due to limitations in relation to the thread safety and re-entrancy of the VS.NET shell we were not successful in this approach. Instead, by separating these parts of the debugger into a local component the threading issues were effectively eliminated.

Another implementation issue arose because of our use of the debugger automation interface that is provided by Visual Studio. This interface is used to extract the contents of program variables when a process is stopped at a breakpoint. Unlike our UNIX debug server API, which is designed to transfer large data structures efficiently, the automation interface is only able to extract data from a simple object with each call. This means that extracting data from complex objects such as arrays can be very time consuming since repeated calls must be made for each element. To solve this problem our package must expose its own interface to the low-level data access facilities provided by the Debug Engines.

A further consideration was in relation to the need to use AIF in the Guard debugger. If Guard operated exclusively in the WIN32 environment then it would be feasible to perform all comparisons using only the native data format and avoid the overhead incurred by using the AIF library routines. However since we wish to use Guard to compare data between UNIX, WIN32 and .NET systems we must employ an architecture neutral format. Because VS.NET employs its own internal format, data arriving at the Debug Engine from a UNIX debug server must first be converted into this format. Once Guard receives the data via the debugger automation interface it must then be converted back into AIF before being processed by the dataflow interpreter. This results in extra overheads because of the dual conversion but simplifies the implementation since we do not need to modify the dataflow interpreter code. We plan to investigate architectural modifications to Guard that remove the need for the multiple format conversions.

One final issue is in relation to the asynchronous behaviour of programs being debugged under the VS.NET environment. We have occasionally observed situations where one process of a multi-process debug session receives significantly more execution time than the others, particularly in cross-language situations or where VS.NET is controlling both managed (.NET) and un-managed (legacy) code. Since the dataflow architecture employed by Guard is designed to deal with this situation, it is not a serious issue, although it can only do so for a finite time before all its internal buffers become filled. We will be monitoring this situation to see if the problem manifests in later versions of .NET and if strategies need to be incorporated into Guard to deal with the issue.

In spite of these difficulties, the implementation has been fairly smooth and a prototype version of Guard has been produced. Visual Studio is one of the few interactive environments that have been designed with the goal of incorporating third party packages [11,13], and we have demonstrated that this integration is possible. Specifically, we have been successful in incorporating a tool that will support the migration of applications to the new .NET Framework.

7. The ‘Earth’ case study

As discussed in Section 1, porting a code from one platform to another poses significant challenges for the programmer. Many of these challenges are present even when the application is only migrated from one version of a language to another, regardless of whether the platform also changes. In this section we illustrate the power of relative debugging using a small scientific program called ‘Earth’, written in Visual Basic. The problems we experienced in doing this were no different from problems we have experienced in the past when we moved code from one platform and operating system to another.

‘Earth’ is a free program that uses the VSOP87 planetary theory to compute the heliocentric ecliptic longitude (L), latitude (B) and the distance to the sun (R) of the planet Earth over a period of several thousands of years [22]. It is based on the same mathematical formulations used to compute long-term, high-precision, heliocentric orbital positions of the planets when preparing astronomical almanacs. Originally written in Visual Basic 5 and Visual Basic 6, we decided to upgrade it to Visual Basic.NET. Visual Basic 6 and Visual Basic.NET have minor, but subtle, syntactic and semantic differences. Whilst the example is performed on the same operating system, it highlights many of the same issues that arise when a program is moved across platforms, and serves as an adequate illustration of how Guard can be applied. The program was converted using a special wizard

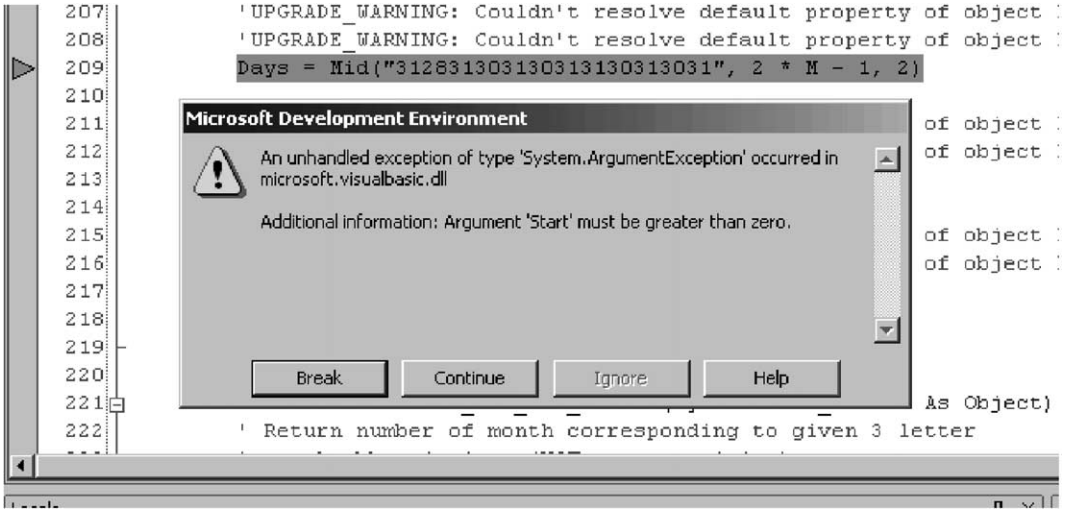


Fig. 4. Exception generated in Earth.

provided by Microsoft, and whilst there were a number of issues that required inspection, the wizard seemed to make sensible decisions during the translation process. In spite of this, the code did not run correctly and consequently required debugging.

7.1. Problem 1

Whilst Earth compiled without errors, it generated a runtime error during execution, as shown in Fig. 4.

Step 1. Using Guard to place an assertion on the statement in error showed that one of the arguments to the `Mid` function was different between the two versions, in particular, the variable `M` had a value of 1 in the Visual Basic 6 code and 0 in the Visual Basic.NET version. Fig. 5 shows the code used in the derivation of `M`.

Step 2. Placing assertions on `Q` and `Month_Year_BCAD` indicated that they were both wrong. Since `Month_Year_BCAD` was passed as an argument, we next inspected the location of the call to `DAYS_IN_MONTH_OF`. Fig. 6 shows the code fragment responsible.

Step 3. Placing assertions on `M`, `Y`, `MonthSelect.Text` and `Year_Renamed.Text` indicated that they were all incorrect, more over `MonthSelect.Text` and `Year_Renamed.Text` were actually initialised.

Step 4. Closer inspection of the code showed that there were multiple calls to the function

```
Public Function DAYS_IN_MONTH_OF(ByRef Month_Year_BCAD
      As Object)_ As Object
      Dim Leap, Q, Days, Y As Object
      Dim M As Integer
      Q = Trim(Month_Year_BCAD)
      M = MONTH_NUM_FOR_ABBREV(Left(Q, 3))
      Y = Trim(Mid(Q, 4, 16))
      Days = Mid("312831303130313130313031", 2 * M - 1, 2)
      If IS_LEAP_YEAR(Y) = True And M = 2 Then Days = 29
      DAYS_IN_MONTH_OF = Days
End Function
```

Fig. 5. Code fragment for computing `M`.

```
Private Sub ADJUST_MONTH_LENGTH ()
                  :        :        :
      D = Val(Trim(DaySelect.Text))
      M = Trim(MonthSelect.Text)
      Y = Trim(Year_Renamed.Text)
      If BCOption.Checked = True Then
                  Y = Y & " BC"
            Else
                  Y = Y & " AD"
      End If
      NDays = Val(DAYS_IN_MONTH_OF(M & " " & Y))
```

Fig. 6. Code fragment for calling DAYS_IN_MONTH_OF.

ADJUST_MONTH_LENGTH, some of which were as a result of handling changed field events. Using the Visual Studio Call Stack view, we observed that ADJUST_MONTH_LENGTH had in fact been called from different places in the two versions of the program.

A code search showed that MonthSelect.Text and Year_Renamed.Text were initialised in the form-load event handler, but that the .NET version of the program was invoking a changed event handler as a side effect of initialising the form. This in turn caused the ADJUST_MONTH_LENGTH code to be executed earlier than in the Visual Basic 6 version, and thus before it was properly initialised.

7.1.1. The root cause of this error was different event ordering in Visual Basic 6 and Visual Basic.NET

Even a simple error such as this illustrates the power of relative debugging. Using conventional techniques we would have quickly been able to determine that the value of M was incorrect, since it was generating an invalid value for the Mid function. However at this point we would have had no idea whether the root of the problem arose in the MONTH_NUM_FOR_ABBREV function or in the value of Q. In tracking the cause we might have spent considerable time examining code for errors, even though it was actually correct. Using relative debugging we immediately know where the source of the error is located.

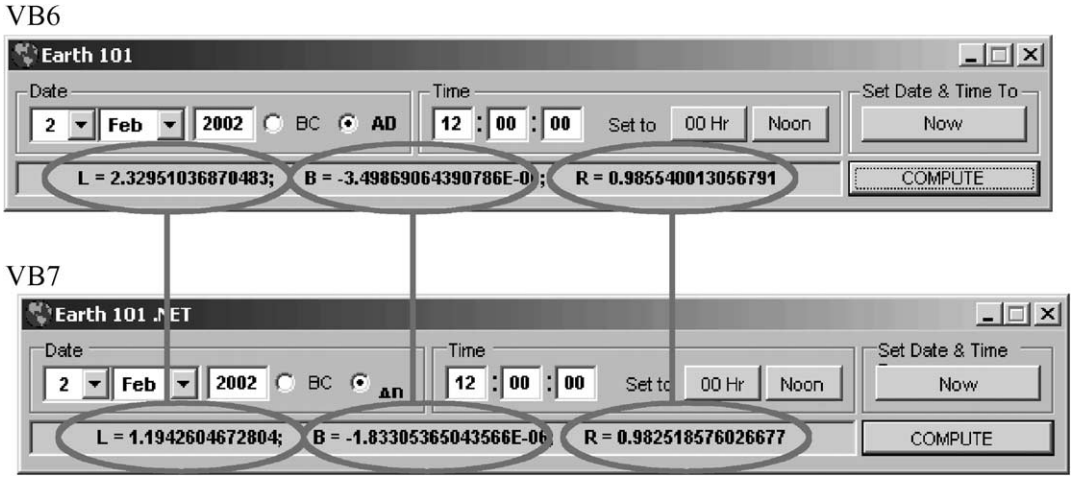


Fig. 7. Values of *L*, *B* and *R* in error.

```
Private Sub ComputeButton_Click(ByVal eventSender As System.Object,
ByVal eventArgs As System.EventArgs) Handles
                                ComputeButton.Click
    Dim Q As Object
    Q = JDE_FOR(INTERFACE_DATE, INTERFACE_TIME)
    Q = EARTH_LBR_FOR(Q)
    Text2.Text = Q
End Sub
```

Fig. 8. Code fragment for computing *L*, *B* and *R*.

```
    Public Function JDE_FOR(ByRef Date_String As Object,
                      ByRef Time_String As Object) As Object
        Dim Q, W As Object
        W = JD_NUM_FOR(Trim(Date_String))
        Q = Trim(Time_String)
        Q = (Val(Left(Q, 2)) * 3600# + Val(Mid(Q, 4, 2)) * 60# +
              Val(Mid(Q, 7, 16))) / 86400#
        JDE_FOR = W + Q
    End Function
```

Fig. 9. Code fragment for JDE_FOR.

```
Public Function JD_NUM_FOR(ByRef Dd_Mmm_Yyyy_BCAD As Object)
                  As Object
      ...
      Date_String = Trim(UCase(Dd_Mmm_Yyyy_BCAD))
      Q = ""
      For Pointer = 1 To Len(Date_String)
            Q1 = Mid(Date_String, Pointer, 1)
            If Q1 <> " " Then Q = Q & Q1
      Next Pointer
      Date_String = Q
      DD = Val(Q)
      Pointer = InStr(1, Q, DD) + Len(DD)
      ...
      MMM = Mid(Q, Pointer, 3): Pointer = Pointer + 3
      MM = Int(1 + ((InStr(1, "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC",
                  MMM) - 1) / 3))
      ...
      JD = DD + Int(367 * (MM + (Q * 12) - 2) / 12)
            + Int(1461 * (YYYY + 4800 - Q) / 4) - 32113
      ...
      JD_NUM_FOR = JD - 0.5

End Function
```

Fig. 10. Code fragments for JD_NUM_FOR.

```
Pointer = InStr(1, Q, DD) + Len(CType(DD, System.String))
```

Fig. 11. Fixing the coding error.

7.2. Problem 2

On correcting the first problem, the program ran without error. However the values being computed were wrong, as shown in Fig. 7.

Step 1. Using Guard to place assertions on `Q`, `INTERFACE_DATE` and `INTERFACE_TIME` (see Fig. 8) we determined that the error was contained in the `JDE_FOR` function (shown in Fig. 9).

Step 2. Placing assertions on `Date_String`, `W` and `Q` indicated that the routine `JD_NUM_FOR` was in error. The relevant code is shown in Fig. 10.

Steps 3–5. Placing assertions on the result of `JD_NUM_FOR` and the define points of variables `MM`, `MMM`, `Pointer`, `Q` and `DD` allowed us to trace back through the expressions and determine that `DD` was correct but `Len(DD)` was incorrect. On closer inspection it transpired that in the Visual Basic 6 version of the code, the Len function only takes a string argument, and thus when it is passed a variant of type double this is first converted to a character string, and then the length of that string is returned. However, in Visual Basic.NET, the Len function takes an object as a parameter and then it returns the length of the object—in this case a value of 8. The code can be corrected by explicitly converting DD to a string as shown in Fig. 11.

7.2.1. The root cause of this error was different handling of variant parameters with built-in functions

An important point with both of these errors, is that whilst a conventional debugger would have allowed us to explore whether values were correct or not, it would also have required us to have a good understanding of the values that were expected and the algorithms being employed. On the other hand, with relative debugging we did not need to have such a mental model, and could focus instead on just comparing one program with another and tracing the define points of variables. When the data structures are complex and large, this becomes a very significant advantage over conventional approaches. Moreover, the errors were located very quickly and efficiently.

8. Future work and conclusions

It is far too early to claim that .NET is a suitable platform for scientific computation since it has only been released for a short time and there are few commercial codes available, and virtually no scientific ones. As discussed in Section 1 we believe that .NET offers a number of potential benefits for large numeric models. However, the execution environment is very different from other platforms and so it is critical that as many tools as possible are available to facilitate the transition of existing legacy software. Guard is one such tool because it allows a user to compare two executing programs simultaneously on different platforms.

Whilst the implementation of Guard under UNIX alone is mature and has been used on many case studies, the current version under VS.NET is still in a pre-beta testing phase. Specifically, the control of programs by Visual Studio under UNIX at this stage is rudimentary and this is why the case study presented here focussed entirely on the Windows operating system. We are also planning a number of extensions that will be required if Guard is to be of practical use in supporting the migration to .NET. The current user interface is fairly simple and must be made more powerful if it is to be applied to large programs. At present only simple data types and arrays are supported. We need to extend this to encompass the range of types found in scientific codes, such as structures and other complex types. Assertions need to be able to be saved and restored when the environment is restarted, and assertions should employ symbolic markers which are independent of the actual numeric line numbers.

We are also planning to integrate Guard into Source Safe [10], Microsoft's equivalent of SCCS [18] or RCS [23] making it possible to compare one version of a program with previous versions automatically. We have already experimented with a version of Guard under UNIX that provides explicit support for parallel programming [24], and we plan to enhance the support for multi-process programs in the Visual Studio version to make it feasible to debug programs in running on a cluster of Windows machines. Finally, we are working on a new version of Guard that attempts to perform much of the generation and refinement of assertions automatically. Whilst this project is in the early stages, it appears that using powerful data flow analysis of the two programs would allow us to generate and refine a number of the assertions without user involvement.

Acknowledgements

This work has been funded by grants from the Australian Research Council and Microsoft Corporation. We wish to acknowledge the support of a number of individuals at Microsoft for their assistance on various issues related to Visual Studio and .NET. Particular thanks go to Todd Needham, Dan Fay and Frank Gocinski. We also wish to acknowledge our colleagues, Professor Christine Mingins, Professor Bertrand Meyer and Dr. Damien Watkins for many helpful discussions. Some of the coding for the UNIX debug engine was performed by Le Phu Dung, and the 'Earth' case study was performed by Tim Ka-chung Ho and Clement Chu.

References

[1] D. Abramson, I. Foster, J. Michalakes, R. Sosic, Relative debugging: a new paradigm for debugging scientific applications, Commun. Assoc. Comput. Mach. 39 (11) (1996) 67–77.
[2] D. Abramson, I. Foster, J. Michalakes, R. Sosic, Relative debugging and its application to the development of large numerical models, in: Proceedings of the IEEE Supercomputing, San Diego, CA, December 1995.
[3] D. Abramson, R. Sosic, A debugging and testing tool for supporting software evolution, J. Autom. Softw. Eng. 3 (1996) 369–390.
[4] D. Abramson, R. Sosic, A debugging tool for software evolution, in: Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE-95), Toronto, Ont., Canada, July 1995, pp. 206–214;
D. Abramson, R. Sosic, A debugging tool for software evolution, in: Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Ont., Canada, July 1995.
[5] D. Abramson, R. Sosic, G. Watson, Implementation techniques for a parallel relative debugger, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'96), October 20–23, 1996, Boston, MA.
[6] T. Bemmerl, R. Wismüller, On-line distributed debugging on scaleable multicomputer architectures, High Performance Computing and Networking, vol. II: Networking Tools, Lecture Notes in Computer Science, vol. 797, Springer, Berlin, April 1994, pp. 394–400.
[7] H. Casanova, J. Dongarra, NetSolve: a network server for solving computational science problems, Int. J. Supercomput. Appl. High Perform. Comput. 11 (3) (1997) 212–223.
[8] D. Cheng, R. Hood, A portable debugger for parallel and distributed programs, in: Proceedings of the Supercomputing'94, November 1994, pp. 723–732.
[9] J. Czyzyk, J. Owen, S. Wright, Optimization on the internet, OR/MS Today, October 1997.
[10] http://www.seg.org/research/3Dmodel/SEPformat.htmlhttp://msdn.microsoft.com/ssafe/.
[11] http://www.eclipse.org.
[12] http://www.lahey.com/netwtpr1.htm.
[13] http://www.sun.com/forte/.
[14] L. Snyder, A Programmer's Guide to ZPL, MIT Press, Cambridge, MA, 1999.
[15] T.J. LeBlanc, J.M. Mellor-Crummey, Debugging parallel programs with instant replay, IEEE Trans. Computers C-36 4 (1987) 471–482.
[16] J. May, F. Berman, Panorama: a portable extensible parallel debugger, in: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, May 1993, pp. 96–106.
[17] B. Meyer, .NET is coming, IEEE Comput. 34 (8) (2001) 92–97.
[18] Programming Utilities and Libraries', Sun Release 4.1, Sun Microsystems, 1988.
[19] N. Ramsey, D. Hanson, A retargetable debugger, in: Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, ACM, 1992, pp. 22–31.
[20] R. Sosic, D. Abramson, Guard: a relative debugger, Softw. Pract. Exp. 27 (2) (1997) 185–206.
[21] R. Stallman, Debugging with GDB—The GNU Source Level Debugger, 4.12 ed., Free Software Foundation, January 1994.
[22] J. Tanner, A. Mason, http://FreeVBCode.com.
[23] W. Tichy, RCS—a system for version control, Softw. Pract. Exp. 15 (7) (1985) 637–654.
[24] G. Watson, D. Abramson, Relative debugging for data parallel programs: a ZPL case study, IEEE Concurrency 8 (4) (2000) 42–52.
[25] R. Wismuller, M. Oberhuber, J. Krammer, Interactive debugging and performance analysis of massively parallel applications, Parallel Comput. 22 (3) (1996) 415–442.

David Abramson has been involved in computer architecture and high-performance computing research since 1979. Previous to joining Monash University in 1997, he has held appointments at Griffith University, CSIRO, and RMIT. At CSIRO he was the program leader of the Division of Information Technology High-Performance Computing Program, and was also an adjunct Associate Professor at RMIT in Melbourne. He was also a program manager in the Co-operative Research Centre for Intelligent Decisions Systems. Abramson is currently the head of the School of Computer Science and Software Engineering (CSSE) at Monash University, Australia. CSSE consists of over 80 academic staff members across two campuses. He is a project leader in the Co-operative Research Centre for Distributed Systems Nimrod Project and also Chief Investigator on an ARC funded research project called Guard, a relative debugger. Abramson has chaired a number of international conferences, including the prestigious ACM International Symposium on Computer Architecture in 1992. He has published over 100 papers and technical documents. He has given seminars and received awards around Australia and internationally and has received nearly $2 million in research grants. He is a co-founder of Active Tools P/L with Dr. Rok Sosic, a company which was established to commercialise the Nimrod project and Guardsoft, a company focused on commercialising the Guard project. Abramson's current interests are in high-performance computer systems design and software engineering tools for programming parallel and distributed supercomputers.

Greg Watson is a technical staff member in the Advanced Computing Lab at Los Alamos National Laboratory. His research interests focus on tools for parallel computers, program debugging, distributed computing and operating systems. He received his BS in computer science from the University of Tasmania and his PhD from Monash University. Prior to his current position, Greg worked as a senior research fellow in the School of Computer Science and Software Engineering at Monash University. He is a member of the IEEE, Immediate Past President of the Internet Society of Australia and Co-Chair of the Australian Domain Administration.

Parallel Relative Debugging with Dynamic Data Structures

David Abramson †, Raphael Finkel ‡, Donny Kurniawan †, Victor Kowalenko †, Greg Watson §

† School of Computer Science & Software Engineering, Monash University, Clayton Australia, 3800
davida@csse.monash.edu.au

‡ Department of Computer Science University of Kentucky Lexington, Kentucky U.S.A.
raphael@cs.uky.edu

§ Advanced Computing Lab Los Alamos National Laboratory, Los Alamos, NM, 87545, U.S.A.
gwatson@lanl.gov

Abstract

This paper discusses the use of "relative debugging" as a technique for locating errors in a program that has been ported or developed using evolutionary software engineering techniques. It works on the premise that it is possible to find errors by comparing the contents of key data structures at run time between a "working" version and the new code. Previously, our reference implementation of relative debugging, called Guard, only supported comparison of regular data structures like scalars, simple structures and arrays. Recently, we augmented Guard enabling it to compare dynamically allocated structures like linked lists. Such comparisons are complex because the raw values of pointers cannot be compared directly. Here we describe the changes that were required to support dynamic data types. The functionality is illustrated in a small case study, in which a parallel particle code behaves differently as the number of processors is altered.

1 Introduction

In 1994 Abramson and Sosic introduced a new debugging and testing strategy called "Relative Debugging". Relative debugging allows a user to debug a faulty program against a working version, possibly running on another machine, in another language, and on a different operating system. It is particularly valuable when programs are ported from one platform to another, because it allows a user to locate the erroneous region very quickly using a divide-and-conquer strategy. Relative debugging is particularly useful when codes are ported to parallel computers, because subtle errors are often introduced at this stage. We have built a tool called "Guard" that implements the key features of relative debugging in addition to conventional debugging primitives.

Relative debugging works by allowing a user to compare the contents of various data structures between executing codes at particular times. Using this approach, it is possible to determine at which point a new version of the code diverges from an existing "reference" code. We have shown using many different case studies that relative debugging is a powerful technique [1] [2] [3] [4] [5] [12] [14] [16] [15].

To date, our focus has been on comparing the contents of static structures like scalars and arrays. This limitation has been partly because the applications of interest to us, namely scientific codes, make extensive use of static structures. In addition, it is fairly easy to see how to extract and compare array structures between two running codes, because arrays are usually regular, and their representations do not change much between different computer architectures or even languages. In order to cater to applications in which the shape may change as the code evolves, Guard implements a powerful algebraic specification language. This language makes it possible to describe how the data structure has changed, and so the debugger can use this mapping to match up the variables at run time.

Comparing dynamic data structures between programs is significantly more difficult than for static arrays, because we wish to compare the contents of the structures without using absolute values like pointers. In fact, we actually want to build a "normalized" structure independent of the machine and the way it stores dynamic pointers. The approach we have adopted is to first serialize the data structures, which might be arbitrarily large and may include cycles. The result has two parts: a type description and a data description. We then compare the type descriptions for equivalence. If they are equivalent, then we compute the difference between the two data descriptions. That difference is presented to the user.

This paper begins with a discussion of relative debugging, and in particular, its implementation in Guard. We then describe how we have implemented the comparison of dynamic structures, including the extensions to the machine-independent type system we have previously developed called AIF. Finally, we illustrate the new work by applying it to the debugging of a parallelized code based on SPH, a technique that is used widely in solving problems in computational fluid dynamics (CFD). In this case study, the SPH code produced different results when running on parallel machines with different numbers of processors. Guard was able to help us locate the source of this divergence and correct it.

2 Relative Debugging and Guard

A relative debugger like Guard implements a standard set of commands for controlling the execution of a program and for examining the state. In addition, it allows a user to launch and control more than one program at a time, possibly on different physically distributed platforms. Data comparison is facilitated by a Guard command that allows the user to define an *assertion* between data structures. An assertion specifies a pair of tuples, each containing a program name, a data structure identifier and a line number, as shown below:

```
assert prog1::var1@line1 = prog2::var2@line2
```

In this example, the assertion indicates that `var1` and `var2` should have the same values at `line1` in `prog1` and `line2` in `prog2`. In order to implement an assertion, Guard places breakpoints at the relevant line numbers. When the breakpoints are hit, Guard extracts the data from the named variables, and performs a comparison operation. The execution-control mechanism is actually much more complex than described here, and is based on a powerful data-flow interpreter [5]. This mechanism ensures that breakpoint events, which can occur asynchronously and in any order, are managed correctly. Accordingly, each assertion is compiled into a data-flow graph. Guard can support any number of assertions concurrently.

Data structures do not have to be exactly the same. For example, differences in floating-point numbers are compared within a tolerance. Moreover, the data structure types do not need to be identical, as long as Guard knows how to coerce one type into another. Thus, it is possible to compare a `short int` against a `long int`. Arrays can actually be different shapes, as long as there is a unique and regular mapping from every element of one array to another. For example, it is possible to compare a vector with a two-dimensional array as long as the user can describe the transformation between the two structures. This transformation is described using an algebraic specification language, which is then interpreted by Guard as it maps one structure to another [15].

Differences between the contents of structures can be reported using various techniques. For example, for low dimensioned data, or scalars, it is generally adequate to report the values and the differences. For higher ordered data structures like multidimensional arrays, it is possible to use scientific visualization packages like Open DX [8]. Such visualizations can be very helpful in determining why two programs produce different data, because it is possible to highlight the nature of the divergence. Previously, our experience was in visualizing multidimensional floating-point arrays [1][2]. In this paper we use some new visualizations more suited to dynamic data.

3 The Architecture Independent Form (AIF)

Guard is machine and architecture independent. It can extract data from two different machines and manipulate and compare the data on a third. In order to provide this functionality, we have built a machine and language independent representation for data types called Architecture Independent Form (AIF) [15]. AIF represents types using string expressions. These expressions can then be stored and manipulated as simple strings, and the data they represent can be manipulated using a set of portable routines. As a result, we can perform operations on data types that are not supported by the base architecture, for example, adding 64-bit integers on 16-bit processors, or manipulating big-endian quantities on little-endian machines. AIF facilitates the high degree of machine heterogeneity provided by Guard.

3.1 Type descriptions

An AIF representation has two parts, namely the type descriptor and the data descriptor. Type descriptors are represented in ASCII and consist of a single letter followed, if necessary, by modifiers indicating precision and sign and any subordinate type descriptors. Table 1 shows the currently supported set of AIF types. The items above the double lines were supported in previous versions of AIF [5] [15], and the new types below the double line have been added specifically to support dynamic and structured data.

The atomic types are straightforward. The void type, described by the type descriptor `v`, has a 0-length associated data descriptor. A character is always a single byte of data, so its type descriptor, `c`, gives no size information.

Strings are assumed to be null-terminated, so their type descriptor, `s`, also lacks size information. The integer 3 could be represented by type descriptor `is1` (one-byte signed integer), `iu8` (8-byte unsigned integer), or a variety of alternatives. The associated data descriptor has the specified number of bytes with the integer 3 stored in big-endian format. There are no inherent restrictions on precision, although some precisions cannot be converted to the native integer representation on some architectures. Floats are similar, but they are harder to render in a canonical form. We have chosen to store the data bytes in big-endian order but otherwise do not interpret the bits of mantissa and exponent. The result is that floats are not as machine-independent as we would prefer; which needs to be addressed.

Letter	Modifiers	Description
v		Void
c		Character
s		String
i	s or u, *b*	signed/unsigned integer, *b* bytes
<		enumeration
f	*b*	float, *b* bytes
[	*l .. hd*$_1$*]* *d*$_2$	array with bounds *l..h*, index type *d*$_1$, base type *d*$_2$
{	*n*$_1$:*d*$_1$, *n*$_2$:*d*$_2$, …}	record with field *i* named *n*$_i$ of type *d*$_i$
^	*d*	pointer to a component with type descriptor *d*
%	*n/d*	component names *n* with type descriptor *d*
>	*n/*	reference to component named *n*

Table 1 – AIF type descriptions

Arrays are all considered one-dimensional. As in Pascal, multidimensional arrays can be represented as arrays of arrays. The type descriptor starts with `[` and includes two numbers indicating the low and high bounds and two subordinate type descriptors, one for the index type and the other for the base type. The index type must be an integer or enumeration so the bounds make sense. A character array of two elements, containing "a" and "b", could have type descriptor `[0..1is4]c`; the associated data descriptor would be two bytes long and contain the two characters.

Structured types are more complex. Pointers to data are described with the `^` type descriptor, which takes as a modifier a type descriptor for the base type. A pointer to a pointer to the character "a", for instance, has the type descriptor `^^c`. Pointer data types require one byte of data in addition to whatever data the base type requires. The extra byte of data has a special meaning: 0 indicates a null pointer, 1 indicates a non-null pointer, and other values are discussed shortly. In this example, the data descriptor is three bytes long, containing 1, 1, and "a". It does not contain any machine addresses involved in the double indirection, because that information is not germane to the value.

Records have an arbitrary number of named fields. Their type descriptor, which starts with `{`, has an arbitrary number of comma-separated (name, type descriptor) pairs. Names are ASCII strings. The amount of data associated with a record is dependent on the amount of data required by each field.

Data structures can be recursively defined. For example, a linked list of integers has a type that depends on itself: `{a:is4,b:^{a:is4,b:^{+a:is4, …}`. The recursion is broken by introducing a name such as `n1` for the replicated part and referring to it, giving us the compact representation `%n1/{a:is4,b:^>n1/}`. In this type descriptor, the characters `/` and `:` are used as delimiters to end names such as `a`, `b`, and `n1`.

Components can also be named when substructures have the same type. For example, a binary tree might have type descriptor `%n2/{lft:^>n2/,rght:^>n2/}`.

Data itself can be circular when there are pointers. For instance, a linked list such as the one named `n1` above might have one element that points to itself. The special pointer values are extended in order to cover this case. A pointer value of 2 indicates a forward reference, and the subsequent 4 bytes are reserved to hold a reference name. A pointer value of 3 indicates a backward reference in the data stream. Again, the subsequent 4 bytes holds the reference name. A pointer value of 4 indicates an invalid pointer, in which case no data is stored. A linked list of type `n1` with a single node containing 8 and a pointer to itself might be represented by this data stream: 2 (named pointer value), 0, 0, 0, 1 (name), 0, 0, 0, 8 (four bytes for the integer 8), 3 (named pointer reference), 0, 0, 0, 1 (name).

Named data references are also useful when the same values appear in several places within the same data structure. A binary tree with common sub-trees can make use of named data references to reduce the length of its data representation, although such reduction is not necessary to render the data representation finite.

3.2 AIF routines

The AIF library contains conversion routines to construct AIF representations from native data of various types, including constructors that build representations of structured types from subordinate structures. To support circular types, the library provides routines for adding names to existing type descriptors. However, it does not contain methods to search down data structures and discover circularities; that task is performed by other software. A second set of conversion routines turn AIF representations into native representations.

The library includes basic arithmetic routines, such as mixed-mode division, which takes two operands in AIF representation and produces a result in AIF representation. The operands may be arbitrary data structures with compatible type descriptors. All integers, of whatever precision, are considered compatible, for instance, but arrays of 2 elements are not compatible with arrays of 3 elements. Arithmetic makes no sense except for arithmetic component types. However, two routines are more general: Two values may be compared for equality, and a value can be tested for equivalence to zero.

We use a recursive definition of comparison and zero-equivalence for compatible structured types, based on component-wise testing. Finally, the AIF library includes printing routines that present AIF representations in human-readable form.

3.3 Gdb extensions

Guard is implemented using a client-server architecture. The Guard client implements the logic required for relative debugging and communicates with one or more debug servers. Each debug server is responsible for operations that control the end application, such as setting breakpoints and examining variable data. In turn, each Guard debug server is implemented using `gdb` for the core services. `Gdb` is a standard debugger for imperative languages such as C, C++, and Fortran [13]. Compilers for supported languages produce debugging information in their executable files that `gdb` can interpret to associate source-line numbers with execution points and names and types with values. We have added a few features to `gdb` to make use of the features of the AIF library.

`Gdb` behaviour can be controlled by a set of run time parameters. For example, the user can enter `set print address` to request that `gdb` include the address of every datum that it prints. We have added two new settings.

First, `set print depth n` tells `gdb` to follow pointers to a depth of `n` (default 0) so the user can see the values that pointers reference. We have found that setting the print depth to about 3 makes ordinary program debugging much easier. In this case, any time we print a data structure, we see up to three levels of indirection with a single print command. This `gdb` enhancement is independent of the AIF library.

Second, `set print aif on` (default off) modifies the way `gdb` outputs all values. It uses the AIF library to construct AIF representations of values, going as deeply as necessary, and detecting data circularities as it goes, after which it outputs the type and data descriptors. The data representation is output in hexadecimal. This feature is of no use to the ordinary user, but it is crucial for Guard because Guard only communicates with the debug servers using AIF.

4 Smooth Particle Hydrodynamics Application

Smoothed/Smooth Particle Hydrodynamics, or simply SPH, is a technique that simulates non-axisymmetric fluid flows and particle motion in diverse subject areas such as astrophysics, fluid mechanics, explosion dynamics and geodynamics.

SPH is a particle method, but unlike typical Particle-In-Cell [7][6][9][10][11] methods based on Eulerian and Lagrangian approaches, it does not require a grid for calculating spatial derivatives in partial differential equations. Instead, derivatives are expressed as interpolation formulae involving a kernel function that is evaluated over a set of disordered points referred to as "particles".

The number of particles required to obtain accurate approximations of the interpolation formulae is reduced substantially by choosing a kernel that decreases rapidly with distance. Then the interpolation formulae become finite sums over nearest-neighbour "particles".

Although many types of kernels can be created, analogous to using different schemes in finite difference methods, the most common kernels are either spline-based or Gaussian. The former are more computationally efficient, while the latter provide a physical interpretation of an SPH equation. All kernels, however, are dependent upon an additional parameter h, which is referred to as the resolution.

The problem of determining nearest-neighbour particles can be overcome by dividing the problem into cells and only considering a limited number of cells near the home cell containing the particle of interest. The size of these cells depends upon the resolution. This layout is shown in Figure 1.

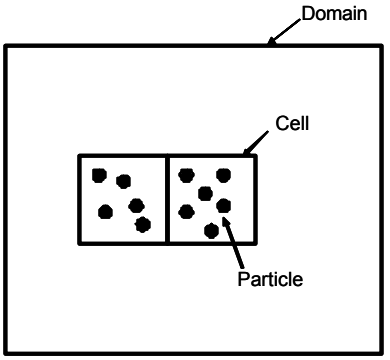
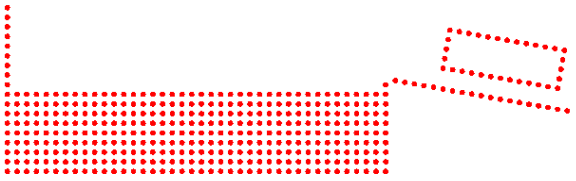


Figure 1 – Layout of Particles and cells in SPH Domain

For a description of the mathematics of the SPH technique, the reader is referred to [10][11].

SPH allows us to simulate a variety of different real world problems. In this paper, we describe the simulation of a fluid flow in a tank containing a solid block, as shown in Figure 2. Figure 2a shows the initial configuration of the block, fluid and particles in the fluid, whereas Figure 2b shows the final configuration. The simulation is able to describe accurately the motion of both the solid block the fluid in the tank.



(a)



(b)

Figure 2 – Initial and Final positions of a block impacting into a tank of fluid, as modeled in SPH

4.1 Implementation issues

The SPH algorithm requires the storage and manipulation of the mass, position, velocity, thermal energy and initial density (sometimes not required) of each particle in the system. Further, every particle must keep track of its neighbours in the same cell. Accordingly, all of the particle properties are stored in a structure, a simplified version of which is shown in Figure 3, and neighbouring structures in the same cell are linked together using a single pointer chain.

```
typedef struct pointst {
        VECTOR          position;
        VECTOR          force;
        VECTOR          velocity;
        double          mass;
        :               :
        :               :
        pointst   *next; }

pointst   *all_particles[MAX_PARTICLES];
```

Figure 3 – SPH Particle Structure in C

The parallel implementation uses the same basic data structures, except the particles are distributed across the processes using a conventional (one-dimensional) domain decomposition; each processor only holds a subset of the particles. A single process controls all of the particles in one grid cell, and the process also controls multiple grid cells. During each time step of the simulation, each process sends the physical data for those particles in the cells located at the boundaries to the adjacent processes. In addition, the processes exchange the datea for those particles that have moved from their home cell to a neigbouring cell. The physical data for the particles in the other processors is stored in ghost bands around the domain.

Figure 4 presents the pseudo code for an SPH computation.

1. **Initialise** positions of particles;
2. **Initialise** the domain decomposition;
3. **While** not end-of-simulation **do**
   a. **Exchange** particle positions with neighbouring processors;
   b. **Perform** the predictor part of the algorithm;
   c. **Compute** cell composition;
   d. **Setup** linked list of particles in cells;
   e. **Perform** the SPH calculation
   f. **Perform** the corrector part of the algorithm;
4. **End loop**

Figure 4 – Pseudo code for SPH calculation

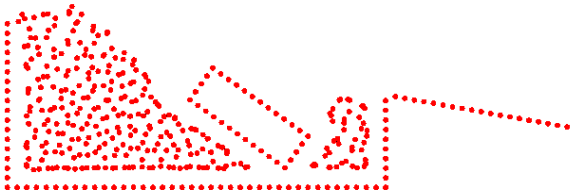
4.2 Debugging SPH

During the parallelization of the original SPH calculation, we noticed that the results differed as we changed the number of processors in the decomposition. The differences were not large, so we attributed them to minor floating-point errors. This type of error is not uncommon when an algorithm is parallelized, because the order of partial sums and other reduction operators can vary, causing slight changes in the computation. Moreover, we did not expect more significant errors, because the parallel algorithm was essentially the same as the sequential one, and the numeric methods were not altered to take account of the parallelization.

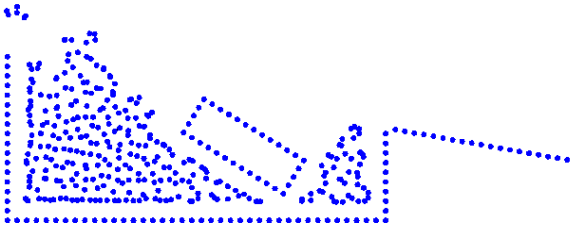
Prior to the enhancements to Guard discussed in Section 4, it would not have been possible to use Guard to trace the source of this discrepancy. The particles are stored in a C structure, and these structures are linked together using a pointer chain. Furthermore, the array of all particles is actually an array of pointers to particle structures. However, the new features in Guard let us not only specify the type declaration, but also perform operations such as COMPARE on two different versions of the program. The normalization of the pointer chains is particularly important, because the virtual address values in the pointers almost certainly differs as the number of processors in the paralle l algorithm is altered.

Our aim was to determine the cause of the divergence. Previous case studies have indicated that programmers have been too eager to attribute different behaviour in parallel programs to the order of reduction operators. In one such case study, we actually found four independent coding errors in the programs that were previously attributed to floating-point differences [14]. Debugging SPH was simply a matter of placing assertions on the key data structures until the source of the error could be located. In addition to assertions, we used scientific visualization techniques to highlight differences in the data. Again, previous case studies [1][2][3][4][14] have demonstrated the power of such visualization.

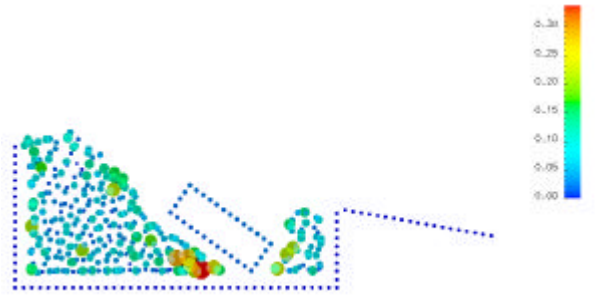
Initially we ran the two programs, one with 2 processors and the other with 4, and placed an assertion on the main data structure representing all_particles at the end of the computation (after step 4 in Figure 4). Figures 5 (a) and 5 (b) show the two different final particle configurations, . Figure 5 (c) highlights the difference by scaling the glyphs for each particle according to the size of the Euclidean distance between corresponding particles in each version of the code. This is a better approach for examining the difference in particle positions.



2-processor version (a)



4-processor version (b)

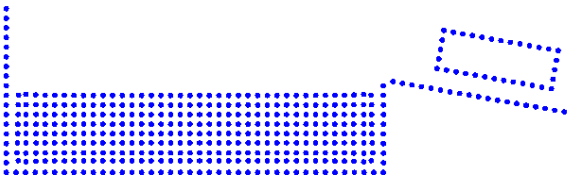


Difference (c)
Figure 5 – Final particle configuration

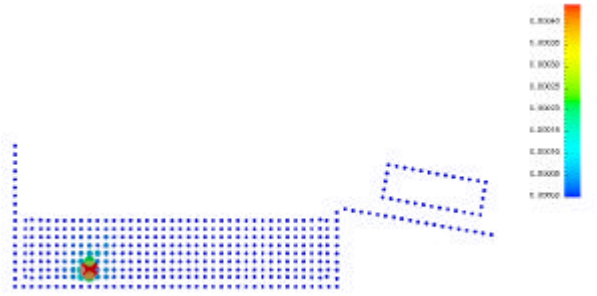
Figure 6 shows the same data but after the first time step only (after step 3f in Figure 4). Although it is extremely difficult to see the difference between the data by examining Figures 6(a) and 6(b), the difference is clearly seen in Figure 6(c).



2-processor version (a)

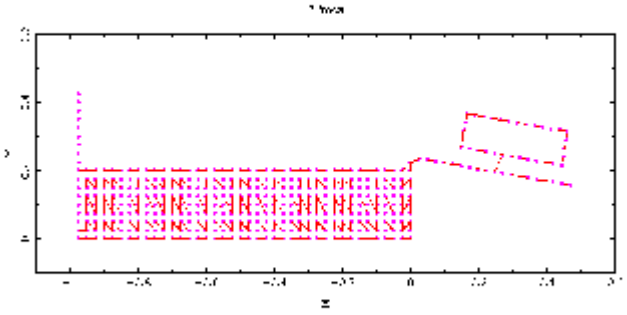


4-processor version (b)

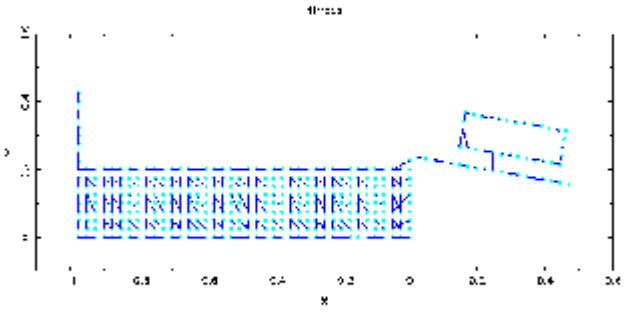


Difference (c)
Figure 6 – Particle configuration after one time step

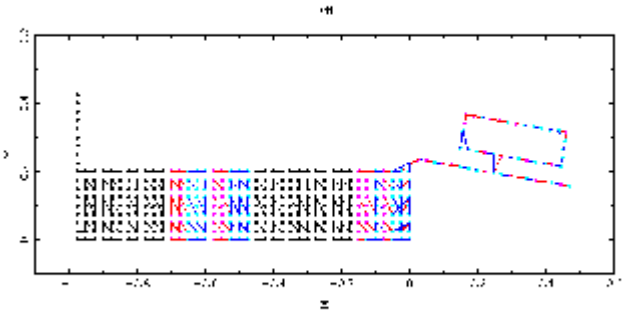
Further assertions helped us refine the region in which the computations diverge. Specifically, Figures 7(a) and 7(b) show the particles and the linked lists that indicate neighbouring particles. In these visualizations, a link is shown by drawing a line from one particle to another. Although it is difficult to see the difference by observing the images, Figure 7(c) highlights the difference by colouring the links black if they are identical in the two codes; otherwise the two different links are superimposed on each other. At this point, the visualisation highlights the problem, because the black regions correspond to particles allocated to processors 0 and 2, whereas processors 1 and 3 have differences. Further examination of the code, indicated that the computation which determined the composition of particle cells is dependent on the number of processors, and thus each version of the code generates a different cell structure. To ensure that the two programs generate identical solutions we modified the calculation so that the cell composition was independent of the actual number of processors. Importantly, Guard was instrumental in localizing the region that was responsible for the different behaviour in the two versions of the code.



2 processor version (a)



4 processor version (b)



Difference (c)

Figure 7 – Inter particle links

5 Conclusions

This paper has described the enhancements to Guard to enable it to represent, manipulate and compare complex dynamic data structures. In particular, we have described the changes required to the Architecture Independent Form (AIF), including normalization of memory address pointers. These enhancements are novel and can be used in other tools that manipulate such dynamic structures.

We have described the application of relative debugging for determining why two different versions of an SPH program generated different results. Prior to performing the case study none of the application developers was able to explain the discrepancy in these results, and were of the belief that the divergence was due to different partial orderings in the parallel computations. However, Guard enabled us to demonstrate that the contents of important data structures were actually different in both versions, and that the difference was due to a single variable, namely the number of processors being used. When we altered this critical computation, we could force the two programs to compute exactly the same result. This outcome is important, because developers frequently attribute floating point issues as the cause for the different behaviour without a rigorous examination of the actual cause Guard is an important tool because of the relative ease with which it can perform the comparisons between different versions of codes.

The paper also highlighted the importance of scientific visualization techniques in showing where the contents of data structures differ. In this work we have presented new visualizations that show small differences in particle position that cannot be detected by simple visual inspection of the particle configurations. Furthermore, we were able to show the dynamic links between particles, and the differences in these links.

At present, it is necessary to produce new visualization scripts for each problem, and this can be time consuming. In the future, we would like to achieve better integration

between Guard and production scientific visualization packages.

Acknowledgements

This work is supported by a research grant from the Australian Research Council. We wish to acknowledge support from the Monash Cluster Computing laboratory (MC^2).

References

[1] Abramson D., Foster, I., Michalakes, J. and Sosic R., "Relative Debugging: A new paradigm for debugging scientific applications", the Communications of the Association for Computing Machinery (CACM), Vol 39, No 11, pp 67 - 77, Nov 1996.

[2] Abramson D., Foster, I., Michalakes, J. and Sosic R., "Relative Debugging and its Application to the Development of Large Numerical Models", Proceedings of IEEE Supercomputing 1995, San Diego, December 95.

[3] Abramson, D.A. and Sosic, R. "A Debugging and Testing Tool for Supporting Software Evolution", Journal of Automated Software Engineering, 3 (1996), pp 369 - 390.

[4] Abramson, D.A. and Sosic, R. "A Debugging Tool for Software Evolution", CASE-95, 7th International Workshop on Computer-Aided Software Engineering, Toronto, Ontario, Canada, July 1995, pp 206 - 214. Also appeared in proceedings of 2nd Working Conference on Reverse Engineering, Toronto, Ontario, Canada, July 1995.

[5] Abramson, D.A., Sosic, R. and Watson, G. "Implementation Techniques for a Parallel Relative Debugger ", International Conference on Parallel Architectures and Compilation Techniques - PACT '96, October 20-23, 1996, Boston, Massachusetts, USA

[6] Benz, W. “Computer Physics Communications”, **48**, 97 (1988).

[7] Gingold, R.A. and Monaghan, J.J. “Monthly Notices of the Royal Astronomical Society”, **181**, 375 (1977).

[8] http://www.opendx.org/.

[9] Lucy, L.B. “Astronomical Journal”, **82**, 1013 (1982).

[10] Monaghan, J.J. “Annual Reviews of Astronomy and Astrophysics”, **30,** 543 (1992).

[11] Monaghan, J.J. “Computer Physics Reports”, **3**, 71 (1985).

[12] Sosic, R. and Abramson, D. A. "Guard: A Relative Debugger", Software Practice and Experience, Vol 27(2), pp 185 – 206 (Feb 1997).

[13] Stallman, R. Debugging with GDB – The GNU Source Level Debugger, Edition 4.12, Free Software Foundation, January 1994.

[14] Watson, G. and Abramson, D. “Relative Debugging For Data Parallel Programs: A ZPL Case Study”, IEEE Concurrency, Vol 8, No 4, October 2000, pp 42 – 52.

[15] Watson, G. R., “The Design and Implementation of a Parallel Relative Debugger”, PhD Thesis, Monash University, Melbourne, October 2000.

[16] Watson, G. and Abramson, D. “The Architecture of a Parallel Relative Debugger”, 13th International Conference on Parallel and Distributed Computing Systems - PDCS 2000, August 8 - 10, 2000.

DUCT: An Interactive Define-Use Chain Navigation Tool for Relative Debugging

Aaron Searle*,1, John Gough*,1,
David Abramson†,2

* *Centre for Information Technology Innovation, Queensland University of Technology, Brisbane, Queensland, Australia*
† *School of Compute Science & Software Engineering, Monash University, Clayton, Victoria, Australia*

ABSTRACT

This paper describes an interactive tool that facilitates following define-use chains in large codes. The motivation for the work is to support "relative debugging", where it is necessary to iteratively refine a set of assertions between different versions of a program. DUCT is novel because it exploits the Microsoft Intermediate Language (MSIL) that underpins the .NET Framework. Accordingly, it works on a wide range of programming languages without any modification. The paper describes the design and implementation of DUCT, and then illustrates its use with a small case study.

KEYWORDS: AADEBUG2003; Relative Debugging

1 Introduction

In 1994 Abramson and Sosic invented a new debugging paradigm called "Relative Debugging" [AFMS96, AFMS95, AS96], and developed a reference implementation called Guard [SA97]. Relative debugging allows a user to both test and debug an application against a working reference version. It supports the idea of software evolution in which a program is changed incrementally, possibly because of changes in the program function, but also because of external changes in the environment, such as different computer architectures, operating system and run time libraries, and languages. Relative debugging facilitates the comparison, at run time, of the contents of key data structures. The methodology supports iteratively refining a program code until the source of a divergence can be discovered. This approach means that even large programs can be reduced quickly to small regions of code that behave differently. A number of case studies have illustrated the power of the approach [AFMS96, AFMS95].

The most flexible method for comparing data structure contents in Guard is implemented by user specified, declarative assertions. These commands state that a particular data structure in one program, at one line, should be the same as another data structure in another program at a different line. The methodology for deciding where to place assertions is built around following the data and control flow of the codes. For example, if the output of a computation is observed to be incorrect, then the user typically refines the assertions to follow the inputs of that computation – typically by finding

[1]E-mail: {a.searle,j.gough}@qut.edu.au
[2]E-mail: davida@csse.monash.edu.au

the definition points of the variables in the right hand side of the computation. Traditionally, Guard has provided no support for this tracing, and has simply relied on the user being able to traverse the code to find these definition points of interest.

The early versions of Guard were command line driven, and thus, required the user to maintain separate windows onto the two source programs and the debugger. Recently, Guard has been integrated into a number of different interactive development environments (IDEs), namely Microsoft's Visual Studio [Mica], IBM's Eclipse [Suna] and Sun's Sun One Studio [Sunb]. All of these offer significant advantages to Guard as they allow the user to maintain two separate source code views concurrently, and even side-by-side in the environment. Further, specifying assertions is easier, because it is possible to point and click to the variables and lines of interest rather than needing to find the line numbers and manually enter this information into an assertion command. The prototype version of Guard under Microsoft' Visual Studio is called VSGuard.

Having integrated Guard into an IDE we were interested in adding support to assist with identifying and navigating the definition points of the data structures, rather than requiring the user to perform this operation manually. This paper introduces a tool, called DUCT, which provides such support by identifying the definitions that reach a particular variable use within a Microsoft .NET program.

DUCT is novel because it only performs interactive analysis on specified variables, making it fast. Further, program analysis is performed directly on the .NET program which allows DUCT to function with any programming language that targets the Microsoft .NET framework. Finally, embedding DUCT in an IDE allows a user to navigate complex programs quickly, making relative debugging significantly easier than in the past. DUCT also allows users to trace data definitions whilst executing the program under debugger control, providing a unique and powerful platform for locating errors.

The paper is organized as follows. Section 2 introduces the concepts surrounding DUCT and the environment in which it has been implemented. Section 3 discusses DUCT's design, problems encountered during development, and possible enhancements. Two examples that highlight the novel features of DUCT and how it is useful when used within the relative debugging framework are illustrated in Section 4.

2 Background

DUCT locates the definitions of a requested variable by constructing use define (UD) chains [ASU86, HS94]. A UD chain is a list of all definitions that reach a particular variable use within a program. By utilizing UD chains programmers can quickly and efficiently locate the origin of a faulty variable.

The majority of UD chain applications generate the exhaustive set of UD chains for a program by using global analysis techniques [ASU86]. DUCT avoids the inherent expense of global analysis by adopting a demand driven approach [DGS97, HRS95]. Accordingly, DUCT only generates, as requested, UD chains for those variables that are of interest to the programmer. A demand driven approach allows DUCT to be efficiently used as time consuming, and possibly redundant, program analysis on the entire program is avoided. In addition, the generated information is cached so that it may be used during the construction of subsequent UD chains.

DUCT generates UD chains for programs that execute within the Microsoft .NET framework and clearly displays the result within the Microsoft Visual Studio environment [Mica]. A .NET program is contained within a portable executable (PE) file that contains compiler generated Microsoft Intermediate Language (MSIL) [Mic01b] and metadata [Mic01c] that conforms to the Common Type Specification (CTS) [Mic01a]. MSIL is a CPU independent instruction set that can be efficiently transformed into native code (by a just-in-time (JIT) compiler) at runtime. Metadata allows a .NET program to be self-describing by detailing the types, signatures, and other data that the .NET framework uses. DUCT has been integrated into Microsoft Visual Studio .NET using the Visual Studio Integration Program (VSIP) [Micb]. VSIP provides a framework in which the Visual Studio .NET architecture may be extended. The framework allows DUCT to be hosted by Microsoft Visual Studio .NET and

utilize the IDE and services offered by the environment.

Most program analysis and comprehension tools are developed for use with one particular programming language [Sim00, ADS93, DKN01]. DUCT constructs UD chains by performing the required analysis directly on the intermediate code and, therefore, avoids the tight coupling with the high level programming language. Accordingly, DUCT is not bound to one particular programming language, but rather, functions with any programming language that targets the intermediate language.

Recent investigation on static analysis of intermediate languages has focused on traditional global analysis [MS01, SHR+00, Zha00]. However, performing program analysis on intermediate languages for interactive purposes has received little consideration. The work in [LUW99] details the construction of control flow graphs, symbol table information, and UD chain data from Java bytecode.

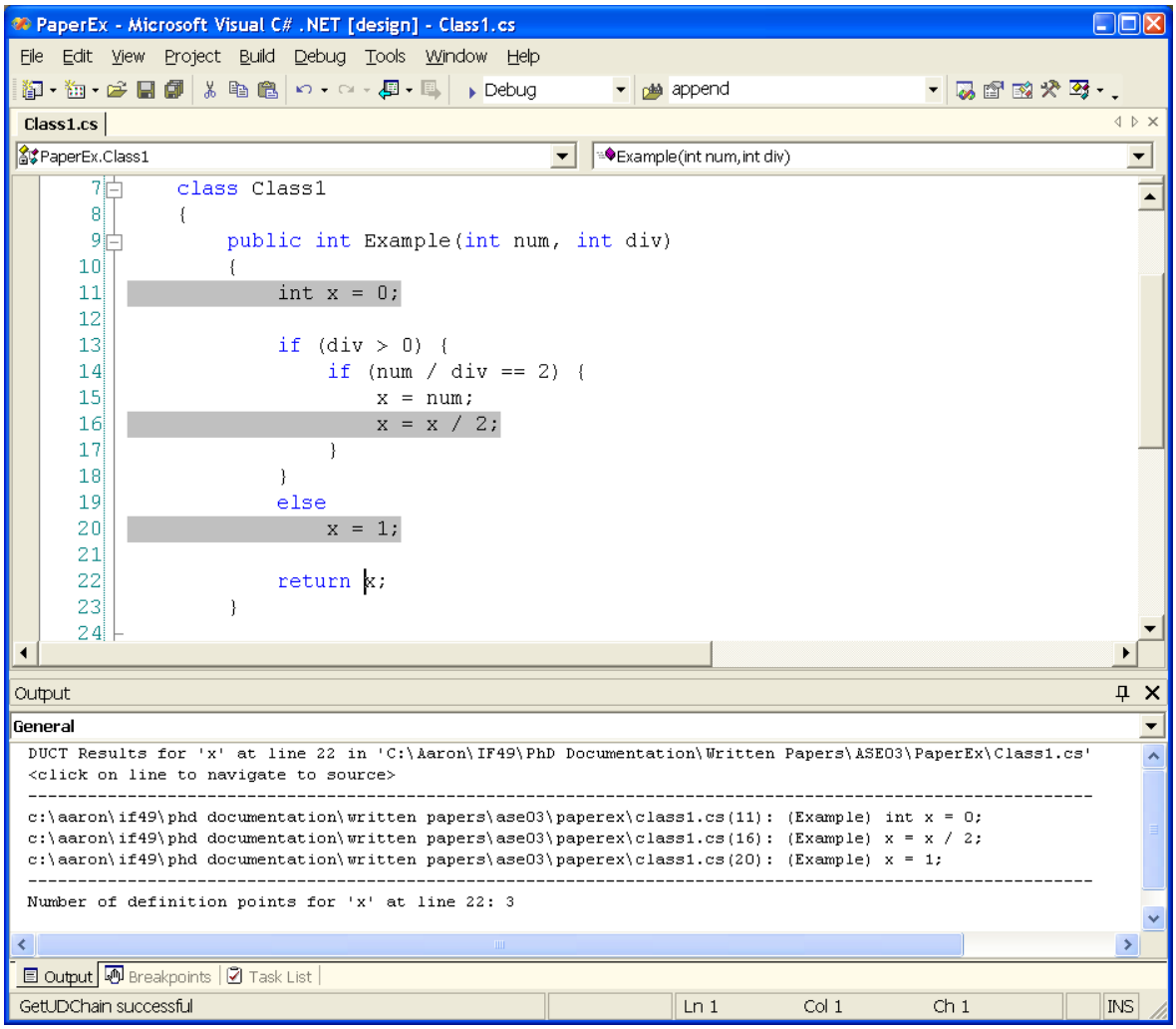


Figure 1: Example illustration of DUCT.

3 DUCT

To locate the definition(s) reaching a variable the user selects the variable in the source program and informs DUCT to construct the UD chain. Currently, the user may select scalar variables, objects, object member fields, and arrays in the local scope for UD chain construction by DUCT. The reaching definition(s) located by DUCT are clearly highlighted in the source window and listed in an output pane that allows the user to easily navigate through the definitions. For example, Figure 1 illustrates the UD chain constructed by DUCT for the variable 'x' on line 22 in a simple C# program. Each definition that reaches the use of 'x' on line 22 is highlighted in the source code window and detailed in the output window at the bottom of the IDE. The user may navigate to a particular definition by clicking on the definition in the output window. This is particularly useful when a definition is not displayed on the screen or is located in another document.

To date, DUCT has been successfully tested with a number of different programming languages, in particular VB.NET, C# and GPCP [QUT].

As discussed, the requested UD chain contains the definitions that reach the selected variable. As DUCT processes the intermediate language, to remain language independent, it cannot rely on the high level language grammar nor can it use the data structures, such as abstract syntax trees, symbol tables or control flow graphs, normally built and maintained by the compiler. DUCT must therefore build the data structures required to construct UD chains by processing the MSIL and metadata emitted by the compiler.

To construct the required data structures DUCT utilizes the metadata and symbolic debug information. The metadata is used to obtain information about types that the code defines and the types that it references externally. The symbolic debug information, stored in the symbol store, contains the information that allows DUCT to interact with the user in relation to the high level source program. The data structures required by DUCT to construct UD chains are summarized in Table 1.

Data Structure	Purpose
Control flow graph	A control flow graph (CFG) provides the information that makes it possible to locate the last definition from each control path that leads to the selected variable. The CFG is constructed using conventional techniques [ASU86].
Call graph	DUCT builds a call graph [HK92] to allow UD chains to be constructed across method boundaries. A call graph represents the possible transfer of control between methods.
Class hierarchy	A class hierarchy [BS96, DGC95] is constructed to allow DUCT to locate all possible definitions when a virtual call is encountered. That is, during UD chain construction each overriding method in each subtype must be scanned for possible variable definitions.

Table 1: Data structures required to construct UD chains.

Instruction	Description
starg.<length> num	Stores a value to the argument numbered num. The value is retrieved from the stack.
stelem.<type>	Stores a value in an element of an array. The array, element index, and value are retrieved from the stack.
stind.<type>	Stores a value of type <type> into memory. The memory address and value are retrieved from the stack.
stfld field	Stores a value into the field field of an object. The object and value are retrieved from the stack.
stloc.<index>	Stores a value into local variable index. The value is retrieved from the stack.

Table 2: MSIL instructions.

Once the required data structures have been constructed, DUCT can locate the variable definitions by identifying MSIL store instructions that update the memory allocated for the selected variable. The MSIL store instructions that DUCT must identify are detailed in Table 2.

When a store instruction is located the target variable must be determined. All store instructions, except for indirect stores, implicitly encode the target variable and, hence, can be extracted directly from the instruction. In contrast, indirect store instructions receive the target address of the variable from the stack. Therefore, to correctly locate the load instruction that pushes the address of the target variable onto the stack DUCT must perform symbolic execution of the abstract stack machine.

To locate the definitions that reach the selected variable the CFG must be traversed backward from the basic block that contains the selected variable. If this block contains a definition for the selected variable (prior to the use) it is the only definition that may reach the selected use (due to the properties of a basic block [ASU86]). If the start block does not contain a definition for the selected variable it is possible that more than one definition may reach the use (along different control paths). Therefore, each control path that leads to the starting block needs to be scanned (backwards) for the last definition. This is achieved by (recursively) scanning the predecessors of the current block.

If a call instruction is encountered (along a path) during UD chain construction and the variable for which the UD chain is being constructed is an object, or passed by reference to the callee, then UD chain construction must continue at the called method. In this case, the required data structures, detailed in Table 1, for the called method are constructed and UD chain construction continues from the last block in the constructed CFG. The control paths in the called method are searched (backwards) for definitions of the corresponding formal parameter.

If each control path in the called method contains a definition for the corresponding formal parameter, then UD chain construction along the current path at the callsite can cease. However, if not every control path through the called method provides a definition for the corresponding formal parameter, UD chain construction must continue along the current control path at the callsite (from the instruction prior to the call).

In addition, if the call is to a virtual function then overriding methods in each subclass need to be processed in the same manner. The constructed class hierarchy is used to determine the overriding methods that are processed in each subclass.

If the beginning of the method has been reached (along a control path) during UD chain construction of a parameter then not all control paths within the method define a value for the incoming parameter. In this case, UD chain construction must continue at each callsite, defined in the constructed call graph, of the current method.

While the demand driven approach provides quick response times an initial, one-off, analysis of the entire program is required to construct the call graph. Specifically, an initial sweep of the program is required to locate MSIL call instructions in order to build the call graph. The construction of the call graph is therefore linear to the size of the program. On the other hand, the class hierarchy can be quickly constructed from the information contained within the metadata and the CFG is only constructed when a method is scanned for definitions.

4 Example

The benefits of using DUCT were highlighted when used to test and debug the Earth program [Cod] as it was upgraded from VB to VB.NET. The Earth program is a free program that uses the VSOP87 planetary theory to compute the heliocentric ecliptic longitude, latitude, and distance to the sun of the planet Earth over a period of several thousand years.

After running the original and upgraded versions with identical inputs we noted, as illustrated in Figure 2, that the resulting output was different. We proceeded to debug the error by locating the code that displayed the erroneous result. The method *ComputeButton_Click*, shown in Figure 3, indicated that the incorrect result being displayed was the value of variable 'Q'. The value of 'Q' is assigned its value by the function call to *EARTH_LBR_FOR* on the previous line. We placed an assertion on the input parameter, 'Q', to determine if the input parameter contained the same value in both versions.

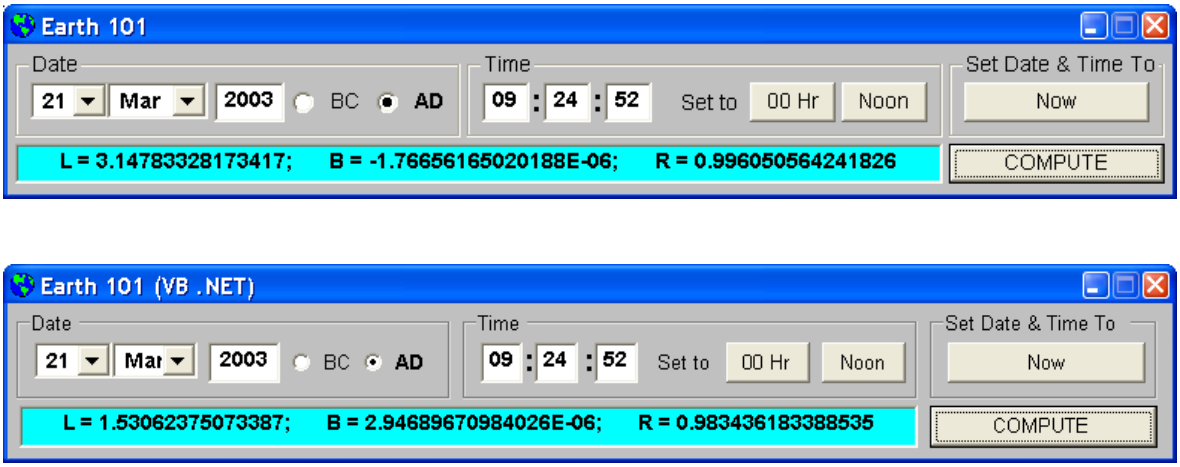


Figure 2: Results displayed from the original and ported Earth programs.

Next, we use DUCT to locate the definition(s) that assign a value to the input parameter 'Q'. Using the result produced by DUCT, illustrated in Figure 3, we discovered that 'Q' was assigned its value by the statement *fracRes* = *W* + *Q* in the method *JDE_FOR* (as shown in Figure 5). We therefore placed assertions on the variables 'W' and 'Q' to determine which variable was influencing the incorrect result.

After setting these assertions we ran VSGuard to determine which variable contained an incorrect value. Guard identified, as shown in Figure 4, that the variable 'W' in the method *JDE_FOR* contained different values in the two programs.

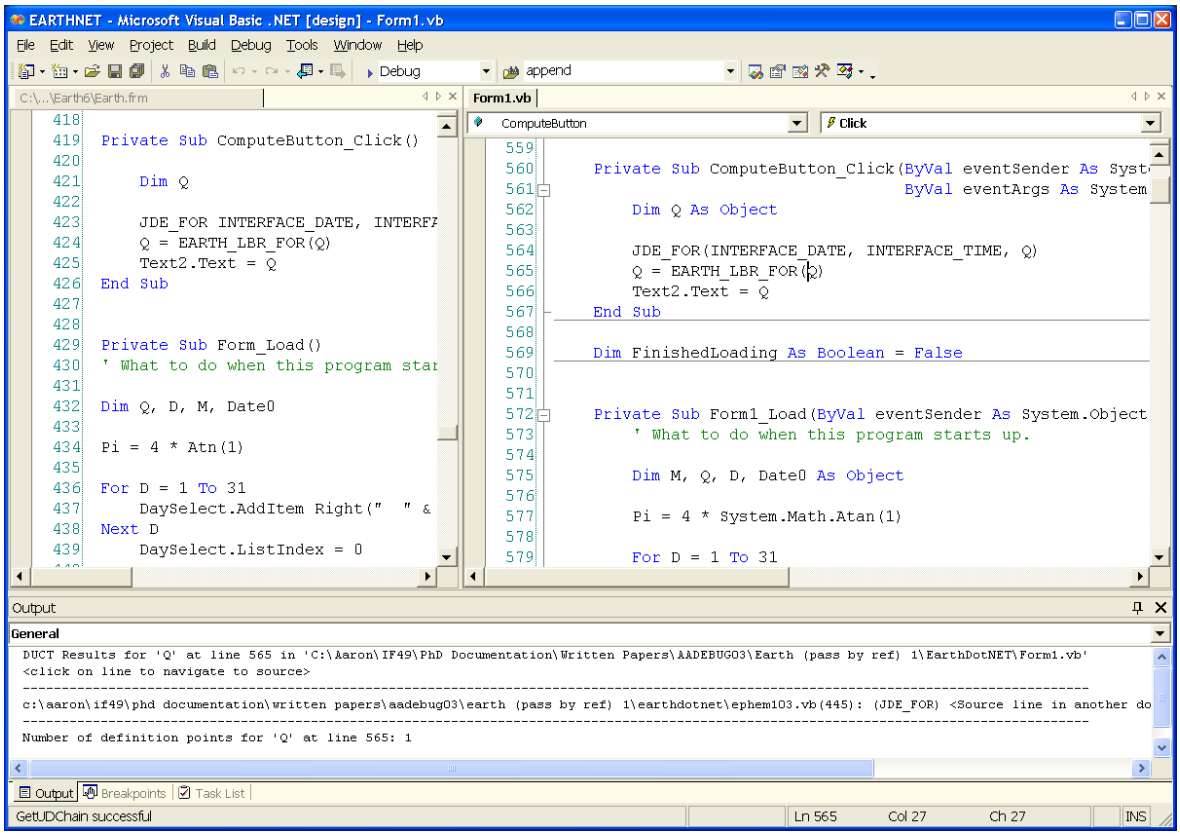


Figure 3: Using DUCT to locate definitions for variable 'Q'.

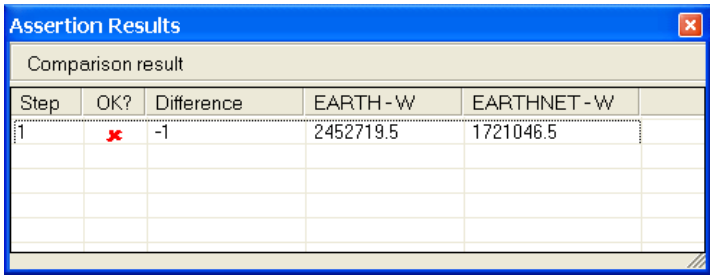


Figure 4: VSGuard results.

Using DUCT again we located the definition that assigns a value to the erroneous variable 'W'. The results produced by DUCT are shown in Figure 5. Navigating to the definition located by DUCT, which resides in the method *JD_NUM_FOR* shown in Figure 6, we continue the process and place assertions on the define points of variables 'MM', 'MMM', 'Pointer', 'Q' and 'DD'. After re-running VSGuard with the redefined assertions we discovered that variable 'DD' was correct but *Len(DD)* was incorrect. It transpired that in the Visual Basic 6 version of the code, the *Len* function only takes a string argument, and thus when it is passed a variant of type double this is first converted to a character string, and the length of that string is returned. However, in Visual Basic .NET, the *Len* function takes an Object as a parameter, and it returns the length of the object - in this case, a value of 8. The code can be corrected by explicitly converting 'DD' to a string as shown in Figure 7.

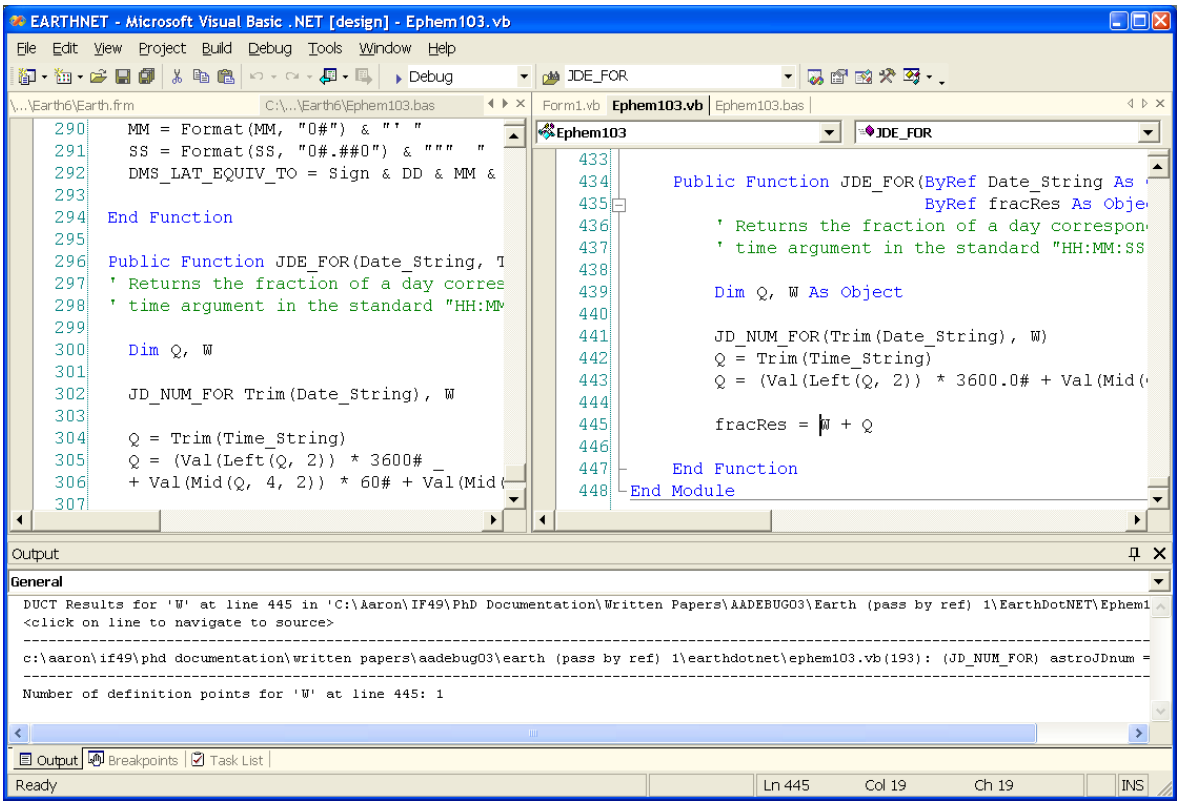


Figure 5: Using DUCT to locate definitions for variable 'W'.

```
Public Function JD_NUM_FOR(ByRef DD_MMM_YYYY_BCAD As Object, _
                           ByRef astroJDnum As Object)
  ...
  Date_String = Trim(UCase(DD_MMM_Yyyy_BCAD))
  Q = ""
  For Pointer = 1 To Len(Date_String)
    Q1 = Mid(Date_String, Pointer, 1)
    If Q1 <> " " Then Q = Q & Q1
  Next Pointer
  Date_String = Q
  DD = Val(Q)
  Pointer = InStr(1, Q, DD) + Len(DD)
  ...
  MMM = Mid(Q, Pointer, 3): Pointer = Pointer + 3
  MM = Int(1 + ((InStr(1, "JANFEBMARAPRMAYJUN-JULAUGSEPOCT ...
  ...
  JD = DD + Int(367 * (MM + (Q * 12) - 2) / 12) + Int(1461 ...
  ...
  astroJDnum = JD - 0.5
End Function
```

Figure 6: Method containing source of error.

```
Pointer = InStr(1, Q, DD) + Len(CType(DD, System.String))
```

Figure 7: Corrected code to eliminate error.

In this example, DUCT was instrumental in following the chain of errors back to the source. It allowed us to navigate the use-define chain quickly, and thus helped us to determine where to place assertions. In all, it only took 4 iterations to locate the error, and a total of 8 assertions were required.

5 Conclusion and Future Work

Relative debugging has been proven to be a powerful paradigm for testing and debugging programs that have undergone software evolution. Relative debugging currently requires the user to identify the data structures and locations within the two programs to define assertion commands. This process can be time consuming and error prone without a detailed knowledge of the programs under consideration. This paper has introduced a tool, DUCT, that allows a user to navigate the data flow of suspect variables throughout the programs. Such information provides valuable assistance when formulating assertion commands.

While DUCT is a valuable tool that assists the user during debugging it also exhibits a number of novel features. DUCT performs efficient demand driven program analysis on the intermediate language. This approach allows DUCT to be used with any high level programming language that targets the intermediate language.

Several limitations and known problems are the subject of ongoing development. Future work will include alias and array analysis. Traditionally, such analysis is global. Investigation will be required to determine an approach that lends itself to the demand driven approach and does not impose severe response times. Safety considerations dictate that any solution is conservative so that no definition point is ever missed.

DUCT does not currently handle delegates (function pointers). Initial investigation suggests that the most efficient way to process delegates is to locate the instantiations of every delegate during the initial program sweep that is required to construct the call graph. Although this approach produces conservative UD chains, we believe the response time will still be adequate.

Instance and virtual methods are passed the instance on which the invocation operates. This parameter is usually referred to as the 'this' parameter in the language community but has various names in high level languages (i.e., 'this' in C#, 'self' in VB, etc). The metadata or symbol store do not contain the high level name that the 'this' parameter is referenced by, preventing DUCT from constructing UD chains for the 'this' object.

DUCT does not construct UD chains for global variables. Difficulties arise, in a multi-threaded program, because the definition(s) for a global variable may reside on different control paths than the one being considered. Investigation is required to determine the best approach to handle global variables.

We are also currently investigating the use of DUCT to automatically generate the assertions between two programs. Minimizing the user's involvement would reduce the cost of maintaining, enhancing, and porting software and have significant impact on current practices in software development.

6 Acknowledgements

This work has been supported by a grant provided by the Australian Research Council under the Large Grants Scheme. We wish to acknowledge the support of Microsoft Corporation who provided

access to the Visual Studio Integration Program (VSIP) whilst it was still in Beta form. Particular thanks go to Dan Fay, Todd Needham and Frank Goscinski of Microsoft.

References

[ADS93] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993.

[AFMS95] David Abramson, Ian Foster, John Michalakes, and Rok Sosic. Relative debugging and its application to the development of large numerical models. In *Proceedings of IEEE Supercomputing*, 1995.

[AFMS96] David Abramson, Ian Foster, John Michalakes, and Rok Socic. Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, 1996.

[AS96] David Abramson and Rok Sosic. A debugging and testing tool for supporting software evolution. *Automated Software Engineering: An International Journal*, 3(3/4):369–390, August 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 324–341, San Jose, CA, October 1996.

[Cod] Free VB Code.
http://freevbcode.com.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, Springer-Verlag, 952:77–101, 1995.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[DKN01] Yunbo Deng, Suraj Kothari, and Yogy Namara. Program slice browser. In *Proceedings of the Ninth International Workshop on Program Comprehension*, Toronto, Canada, May 2001.

[HK92] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering*, Washington, DC, October 1995.

[HS94] Mary J. Harrold and Mary L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Trans. on Programming Languages and Systems*, 16(2):175–204, March 1994.

[LUW99] Don Lance, Roland H. Untch, and Nancy J. Wahl. Bytecode-based java program analysis. In *Proceedings of the 37th Annual ACM Southeast Conference*, Mobile, AL, April 1999.

[Mica] Microsoft. Microsoft Visual Studio .NET.
http://msdn.microsoft.com/vstudio/.

[Micb] Microsoft. Microsoft Visual Studio .NET Integration Program. http://msdn.microsoft.com/vstudio/vsip/.

[Mic01a] Microsoft. ECMA TC39/TG3, Common Language Infrastructure (CLI): Partition I: Concepts and Architecture, 2001.

[Mic01b] Microsoft. ECMA TC39/TG3, Common Language Infrastructure (CLI): Partition III: CIL Instruction Set, 2001.

[Mic01c] Microsoft. TC39/TG3, Common Language Infrastructure (CLI): Partition II: Metadata Definition and Semantics, 2001.

[MS01] Paolo Maggi and Riccardo Sisto. Using data flow analysis to infer type information in java bytecode. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, November 2001.

[QUT] QUT. Gardens point component pascal. http://www2.fit.qut.edu.au/CompSci/PLAS/ComponentPascal/.

[SA97] Rok Sosic and David Abramson. Guard: A relative debugger. *Software - Practice and Experience*, 27(2):185–206, 1997.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

[Sim00] Susan E. Sim. A structured demonstration of program comprehension tools. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, Brisbane, Australia, November 2000.

[Suna] Sun. Eclipse. http://www.eclipse.org/.

[Sunb] Sun. Sun ONE Studio. http://www.sun.com/software/sundev/index.html.

[Zha00] Jianjun Zhao. Dependence analysis of java bytecode. In *Proceedings of the 24th IEEE Annual International Computer Software and Applications Conference*, pages 486–491, Taipei, Taiwan, October 2000.

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:1157–1183
Published online 7 July 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.932

Relative debugging in an integrated development environment

David Abramson[1,*,†], Clement Chu[1], Donny Kurniawan[1] and Aaron Searle[2]

[1]*Faculty of Information Technology, Monash University, Clayton, Vic. 3800, Australia*
[2]*Faculty of Information Technology, Queensland University of Technology, Gardens Point, Brisbane, Australia*

SUMMARY

Relative Debugging allows a user to compare the internal state of two programs as they run, making it possible to test whether two programs perform the same function given the same input. When implemented with a command line user interface, a relative debugger looks like traditional debugging tools with the addition of commands that describe which structures should be equivalent in the two programs. In this paper, we discuss relative debugging within an integrated development environment, and show that there are significant advantages over a command line form. We describe a pluggable, modular, architecture that works with a variety of different products, including Microsoft's Visual Studio, SUN's NetBeans, and IBM's Eclipse. Copyright © 2009 John Wiley & Sons, Ltd.

Received 16 June 2008; Revised 23 April 2009; Accepted 2 May 2009

KEY WORDS: debugging; integrated development environments; software tools

1. INTRODUCTION

It is not uncommon to have multiple versions of a program—one that 'works' and one that is 'broken'. A new version may not work correctly because the programmer changed some code. These errors can often be quite difficult to find, because a change in one place may have an effect somewhere completely different. Alternatively, something may have changed in the environment.

*Correspondence to: David Abramson, Faculty of Information Technology, Monash University, Clayton, Vic. 3800, Australia.
†E-mail: david.abramson@infotech.monash.edu.au

Contract/grant sponsor: Australian Government
Contract/grant sponsor: Object Technology International
Contract/grant sponsor: Microsoft
Contract/grant sponsor: IBM
Contract/grant sponsor: SUN

Copyright © 2009 John Wiley & Sons, Ltd.

For example, the operating system or compilers may have been upgraded, and exactly the same source code may produce different results. Finally, the program may work correctly on one platform, but fail on another. This could occur because of differences in the hardware, compilers, operating systems, or any number of differences between the two platforms. Finding such bugs can be tedious and error prone.

Relative debugging is a technique with which it is possible to compare the internal state of two executing programs [1]. It is an amalgam of *test* and *debugging* methodologies. It enables a user to *test* whether two programs perform the same task by comparing the output of the two programs given the same input. However, it goes beyond traditional test tools by facilitating the comparison of arbitrary data structures at arbitrary points in the execution of the two codes. Thus, it supports *debugging* of a new version of the program by helping to identify the point at which the internal states of the two codes diverge. Relative debugging is a powerful technique for refining a region where a program first produces incorrect results. A programmer does this by following the uses of key variables from their use points back to their definition points, in an attempt to isolate the region of code in error.

Initially, we implemented a relative debugger, called Guard, using a traditional command line user interface. To the user, Guard looks like standard debugging tools such as *gdb*, *dbx*, etc. In fact, many of the commands available in existing debuggers are also provided in Guard. On top of these, Guard adds a few key commands for performing relative debugging—typically commands that describe the structures that should be equivalent in the two programs. However, while powerful, we believe there are significant advantages in using powerful graphical user interfaces (GUIs) such as those exposed in integrated development environments (IDEs). In this paper, we discuss relative debugging in an IDE, and show that there are significant advantages to this. We describe a pluggable, modular, architecture that works with a variety of different products, including Microsoft's Visual Studio, SUN's NetBeans, and IBM's Eclipse.

The paper begins by introducing relative debugging, followed by a discussion of various commercial IDEs, and highlights their functionality. The material provided in Section 2 is by way of background, and is not intended to serve as a single source of information on relative debugging. Indeed, the major contribution of this paper is the design, implementation, and demonstration of relative debugging in an IDE. We then discuss some generic advantages that an IDE offers relative debugging, followed by a discussion of the architecture that we developed. In particular, an IDE can automate some of the more tedious and error prone aspects of the previous implementations. We then show how the various features of a relative debugger in an IDE are implemented in one or more of the IDEs.

2. RELATIVE DEBUGGING

Programs evolve for a large number of reasons. In many situations, large portions of the modified program should produce the same results as the existing version. For instance, if a program is:

- enhanced with new functionality, the existing functionality should remain the same;
- modified to remove an error, the existing unrelated functions should produce the same results. That is, the removal of an error should not introduce new errors;
- ported from one platform and/or language to another, the functionality in the ported program should remain the same.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

In these cases, it is common that the unaffected functionality or program code in the development version will contain data structures that should contain the same values as the original version when the two programs are run with similar inputs.

Relative debugging exploits the information available in previous versions to allow the contents of key data structures to be compared. This allows the user to locate errors by identifying data structures that contain different values when the two programs are executed.

The concept of relative debugging is both language and machine independent. This allows a user to concentrate on the cause of the error rather than the underlying implementation details. The results of several case studies have been reported and show that relative debugging is an extremely effective technique that allows users to locate errors in programs that have been modified or ported to another platform and/or language. In one case study we compared the execution of two large climate models, and were able to isolate errors that were introduced during the parallelization of the algorithms [2]. In this example, subtle semantic errors occurred when the code was modified to allow it to run in parallel, and while it was originally believed that these would not affect the output, our experiments not only illustrated that they did cause different results to be computed, but also helped us locate the source. In another study, we isolated errors in a complex photochemical pollution program as it was ported from one platform to another [1]. In this example, relative debugging allowed us to isolate significant differences in output that were caused by very minor numerical differences in some external library functions (transcendental), and actually highlighted a much more significant error in the overall numerical stability of the code. The technique has also been used to find errors in a nuclear transport kernel as it was translated from one language (C) to an experimental parallel language (ZPL) [3]. This example was interesting because it isolated four errors in the codes—three of which were syntactic errors in the new program, and one of which was incorrect mathematics in the *original* C program. The latter error had subsequently been corrected in the new ZPL program. The case study also demonstrated that relative debugging is effective even when the two programs are written in different languages and with different programming paradigms. Many of these examples concerned quite large, complex, scientific codes. For example, the MM5 weather model used in [2] consists of around 300 000–400 000 lines of Fortran code. In addition to various research projects, relative debugging has been deployed commercially [4], and was recently adopted by a major supercomputer vendor to aid productivity in the development of supercomputing applications [5].

Figure 1 illustrates the relative debugging paradigm using a very simple example. While this program is trivial, it allows us to illustrate the way the technique works. The program on the left depicts the original program, written in Visual Basic, containing a function that computes the factorial for a given number. The program on the right represents the later development version that has been rewritten in C++ and uses a different algorithm to compute the factorial. Despite these differences, the two programs should compute the same factorial value for the given number. Relative debugging may be used to compare the factorial value computed by each program. If the comparison fails, the error can be attributed to the modified factorial function in the development version. In this case, the user may use traditional debugging techniques or continue to apply relative debugging to localize the source of the error even further.

Relative debugging requires the user to identify the corresponding data structures and program points at which comparison should be made within the two programs when they are concurrently executed. The choice of data structures and program points must be determined by the user, based on some knowledge about the two programs. It is not necessary to test all data structures, but only

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

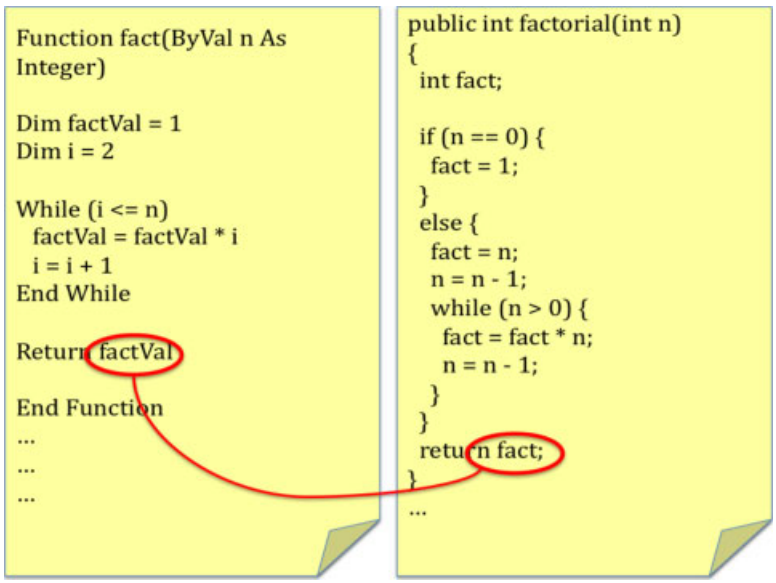


Figure 1. The relative debugging paradigm.

those that help uncover the source of the error. The user provides the comparison details by defining one of the following two types of assertions: imperative assertions and declarative assertions.

2.1. Imperative assertions

An imperative assertion allows the user to manually invoke the comparison of key data structures when the two programs under consideration are suspended. Typically, the user creates a number of breakpoints, at key locations, in the two programs before they are simultaneously executed. When the breakpoints are reached and the programs are subsequently suspended, the user can issue a compare command to compare the contents of key data structures. For example, the following compare command compares the variable factVal in the program orig, with the variable fact in the program dev.

$$\text{compare } \$\text{orig}::\text{factVal} = \$\text{dev}::\text{fact}$$

The user may perform a number of comparisons and gain insight into the error by identifying variables that contain different values across the two programs. To identify where the two programs first begin to differ and isolate the source of the error, the user can define additional breakpoints, resume or restart the two programs, and continue to compare the contents of key data structures.

This process can become tedious and error prone if the user needs to iterate through this process a number of times. The process also becomes laborious if the user wishes to compare the results of a computation that occur within a loop. In this case, the user must manually invoke the comparison each time an iteration is performed.

Copyright © 2009 John Wiley & Sons, Ltd.

2.2. Declarative assertions

A more flexible and less tedious method for defining the comparison details of corresponding data structures is specified by a user-supplied declarative assertion. The declarative approach allows the user to assert that two data structures should contain the same values at particular locations in the two programs.

Declarative assertions are defined before the programs are executed and state that a data structure at a certain line in the original program should contain the same value as the corresponding data structure, at a specific line, in the development program.

Declarative assertions are defined by the assert command. The following assertion statement, for example, states that the variable factVal, in line 114, in the original program should be compared with the variable fact, in line 121, in the development program.

$$\text{assert } \$\text{orig} :: \text{factVal}@114 = \$\text{dev} :: \text{fact}@121$$

The additional line information provided with declarative assertions allows the creation of breakpoints and the comparison of variables to be automated by the relative debugger. In particular, the relative debugger creates a breakpoint in both the original and development versions, for each assertion, at the specified lines. When these breakpoints are reached during execution of the two programs, the relative debugger automatically compares the variables specified by the assertion. If the comparison does not detect an error, the programs are automatically resumed without any user interaction.

Declarative assertions provide a convenient mechanism for stating a set of conditions that must be satisfied if the development version is to be deemed correct. Hence, not only are they useful for isolating the source of an error, but they can also be used to automatically test a new program version against a previous version.

Declarative assertions are also effective if the user wishes to compare the results of a computation that occur within a loop. Unlike the imperative approach, the comparison is automatically performed each time the loop is iterated. As the user is not involved until an error is detected, little user interaction is required to actually detect the presence of an error.

2.3. A systematic approach to relative debugging

Once the user has defined the assertions, the programs are executed and the values of corresponding data structures are compared at the specified locations. If they are run on different platforms they can be executed concurrently, thereby overlapping the two computations and saving time. If a comparison fails, execution is halted and the difference is reported to the user. In the case of an error, the user normally repeats the process of program comparison, with additional or refined assertions, in order to identify the point where values first begin to differ.

To identify the point where values first begin to differ, the user typically formulates additional assertions based on the variables used in the computation of the erroneous value. The methodology for deciding where to place assertions is based on following the data and control flow of the two programs. For example, if the output of a computation is reported to be incorrect, the user would define additional assertions to follow the inputs to the faulty computation—typically by finding the definition points of the variables used in the computation. This process usually continues by a

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

process of refinement until the faulty section of code is localized after which traditional debugging techniques can be used to correct the error.

This systematic approach allows users to locate an error by iteratively defining assertions until the erroneous region is small enough to inspect manually. The technique is extremely efficient because the amount of code that needs to be considered is dramatically reduced after each refinement step. Hence, even in large programs, the user can localize the error to a small region of code with a limited number of refinement iterations.

2.4. Approaches to automatic debugging

There have been attempts over the years to automate the debugging process fully. In 1991, Agrawal stated that '*automatic debugging has been investigated but never really achieved*' [6]. A review in 1988 had already noted that 'most automatic debugging tools were currently only prototypes and did not scale easily to large systems' [7], mainly due to the difficulty involved with capturing the programmer's intuition.

Since the early eighties, a number of research projects and experiments that adopt novel approaches for automatic debugging have been conducted and have provided promising results. These approaches rely on the correctness of the earlier versions of a program to assist the programmer's understanding of the development system and to help with the isolation of faults in the development versions. The approaches attempt either to localize faults or to reduce the search area to a minimal one that allows the errors to be reproduced.

Shapiro proposed *diagnostic algorithms* [8] to identify procedures or functions that behave incorrectly. He proposed an approach known as *algorithmic debugging*, which is a semi-automatic technique where the user is required to answer questions concerning the intermediate results of procedures within the program.

He explains that the 'approach is geared toward languages in which the basic computation mechanism is a procedure (or function) call, but is insensitive to the inner workings of procedures. The diagnosis algorithms abstract away all the details of the computation, except the procedure calls performed, their inputs, and their outputs.' [8].

In 1994 Whalley presented a tool, *vpoiso* [9], that automatically localizes errors in the *vpo* compiler system. The tool isolates optimization errors by determining the first transformation that causes the compiled program to produce incorrect results. Further, vpoiso determines a minimal input that causes the compiled program to produce the incorrect result. Although the tool cannot isolate non-optimizing errors, the minimal input narrows the scope of the problem thereby assisting the developer to localize the error.

DynaDiff [10] is a testing and debugging tool that uses path profiling to isolate potential faults. The tool runs a program twice, with different inputs, and compares the executed control path. The user is informed that an error may exist if an alternate control path is taken or if a control path is executed at different number of times.

Delta Debugging [11] attempts to isolate the cause of an error without user involvement altogether. The technique can be used on any artifact or *circumstance* that influences the execution of a program. Example circumstances are program input, user interaction, and the actual program code.

The Delta Debugging algorithm uses systematic testing to automatically isolate *failure-inducing circumstances*. A failure-inducing circumstance is a subset of the original circumstance that contains the minimal number of elements that cause the original error to re-occur. Each element in the

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

failure—inducing circumstance must be present in order for the error to occur—removing a single element eliminates the occurrence of the failure. Removing the elements that do not influence or impact the failure allows the user to concentrate on the cause of the error.

To identify the failure-inducing circumstances, Delta Debugging systematically removes elements from the original circumstance and re-executes the program to determine if the error remains. This process continues in an iterative fashion until the original circumstance is decomposed to the subset of minimal circumstances that re-produce the original error.

A similar approach to Delta Debugging has been adopted by Renieris and Reiss [12], but they attempt to address the assumption that similar inputs result in similar runs. To overcome this limitation they select a successful run, from a repository of program runs, that mostly resembles the faulty run. The selected successful run is then executed and compared with the execution of the faulty run. The difference of these runs is then reported to the user to assist with locating the source of the error.

This difference also eliminates the assumption imposed by Delta Debugging that similar inputs result in similar runs. WHITHER attempts to address this assumption by selecting a successful run, from a repository of program runs, that mostly resembles the faulty run. The comparison approach using potential invariants, adopted by WHITHER, is similar to relative debugging because it compares a variable from the faulty run with the expected value in the successful run. However, the approach compares arbitrary variables based on the successful runs and assumes that the variables in the successful run should be the same for all runs in the faulty run. Furthermore, the compared variables may not actually influence the faulty program run. Inversely, the approach does not consider situations where the compared variables should actually differ between the two program runs.

In [13], Zeller states '*All in all, relative debugging exploits the existence of a reference run in a classical interactive debugging session. The more of the state and the run is covered by relative assertions, the easier it will be to catch infections early; best results are achieved when porting an otherwise identical program from one environment to another*'.

In comparison with the techniques discussed here, relative debugging is not fully automatic, and still involves the programmer in deciding where to place assertions and in using their intuition in finding errors. The goal of this paper is to illustrate that IDEs play a role in exposing these mechanisms to the user.

3. INTEGRATED DEVELOPMENT ENVIRONMENTS

3.1. Functionality

Developing an application and supporting the full software development life cycle often require a variety of programming tools from different tool vendors. A closer integration of these tools could expedite and streamline the development process [14]. This observation leads to the creation of an IDE. The first IDE was arguably built for the BASIC programming language [15], and was a text-based system that seamlessly integrated code editing, compilation, and program execution [16].

An IDE combines various programming tools such as a source code editor, a compiler, and a debugger into one cohesive environment. The integration allows the tools to work together

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

seamlessly; for example, the debugger can highlight source code errors in the editor and the compiler can be invoked automatically when the code changes. An IDE is a type of computer-aided software engineering (CASE) tool [17], and encompasses a wide range of tools and methods that are used to support software engineering activities such as requirement analysis, system modeling, and testing [18].

IDEs often have a steep learning curve and can be difficult to master [19,20]. However, several studies have shown that once a particular development environment has been mastered, it can significantly increase programmers' productivity [21–25]. IDEs are effective because they improve productivity in three areas, namely efficiency [21], management control, and quality [25]:

- Efficiency: IDEs provide various automation functions, such as refactoring and code-generation support, that can substantially reduce the amount of time required to develop an application. IDEs also promote efficiency by providing development tools that have consistent and cohesive user interfaces.
- Management control: IDEs equip programmers with better control over the development process through features such as automatic recompilation and source code tracking in the debugging phase. In addition, tools such as class browsers and method flow tracers simplify programmers' task in comprehending the source code.
- Quality: IDEs improve code quality by providing tools such as an automatic syntax checker, an integrated help system, and a source code documentation generator. IDE utilities such as unit testers, profilers, and program analyzers assist programmers in detecting code defects at the early stage of development.

3.2. Popular IDEs

3.2.1. Visual Studio

Visual Studio is a suite of application development tools from Microsoft for the Windows platforms on servers, workstations, personal digital assistants (PDAs), and smartphones [26]. Visual Studio can also be used to create applications that target Microsoft Office, the .NET framework, and ASP.NET web framework. It supports several programming languages such as C, C++, C#, J#, ASP.NET, and Visual Basic.NET. Additional third party extensions are available to support other languages. Visual Studio includes various Windows software development kits (SDKs) and it utilizes Microsoft compilers and debuggers.

Microsoft also provides Visual Studio SDK for Visual Studio Industry Partner (VSIP) members that can be used to extend Visual Studio with additional functions [27]. The core foundation of the SDK is the Visual Studio Shell that provides a base IDE that can host custom tools and programming languages. The Shell comes in two modes of operation:

- Integrated mode: Applications built on the integrated Shell will be hosted in the Visual Studio IDE. The mode is used if a vendor wants to provide additional functions in the IDE, for example, support for new programming languages.
- Isolated mode: Applications built with the isolated Shell will be unique stand-alone programs. The mode is used if a vendor wants to develop a specialized tool that does not interact with the Visual Studio IDE, but still leverages the Shell's features.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

3.2.2. NetBeans

NetBeans from Sun Microsystems refers to two distinct products: a platform and an IDE. The NetBeans Platform is a software framework for building cross-platform Java desktop applications [28]. It facilitates the development of rich client applications by providing a number of advanced features such as

- User interface management: The Platform provides high-level presentation components, for example windows, menus, toolbars, editors, palettes, and wizards. It results in standard and consistent user interfaces for rich client applications.
- Data and presentation management: The NetBeans Platform contains a rich toolset for data manipulation and presentation.
- Setting management: A simple and transparent mechanism for saving and restoring user settings and preferences is provided by the Platform.
- Storage management: The Platform offers an abstraction of file-based data access. It provides a common set of APIs to access files regardless of where they are stored. A file object in the NetBeans paradigm may be physically stored on an FTP server, a local disk, or in a database.
- Additional components: The NetBeans Platform has been designed in a modular way. Thus, it is possible to extend the Platform with additional functions required by the rich client applications. Available extensions include specialized editors, remote data access, and automatic network update support.

The NetBeans IDE is an IDE written in Java and built on top of the NetBeans Platform [29]. It can be used to create desktop, web, mobile, and NetBeans-based rich client applications. It has a module-based architecture and the IDE can be extended with various packs. A NetBeans IDE pack is a collection of related modules to support a new feature. Some examples of the packs are: C/C++ Pack for developing C/C++ applications, Enterprise Pack for writing service-oriented architecture (SOA) applications, and Ruby Pack for programming in the Ruby language. Other notable IDE features are:

- A NetBeans Swing GUI builder for developing Java applications with the Swing toolkit [30].
- Project and build systems based on Apache Ant [31].
- An integrated version control system that supports CVS and Subversion [32].
- An extensive support for developing rich client applications based on the NetBeans Platform.

3.2.3. Eclipse

Eclipse refers to both a software platform and an IDE built on top of the Eclipse Platform [33]. Eclipse is written in Java and it runs on multiple platforms where the Java Virtual Machine is available.

The Eclipse Platform defines a set of frameworks and common services that collectively facilitate the creation and integration of IDE-based products and rich client applications [34]. The frameworks and services provide facilities that are required by most tool builders such as [35]:

- A platform runtime system for loading and managing plug-ins.
- A generic windowing system with portable native widget toolkits.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

- Incremental project builders and compilers support.
- A language-independent debug infrastructure.
- Tools and APIs for managing workspace resources such as files, folders, and projects.
- An integrated help system.

The Eclipse Platform is modular and all platform functions are implemented as plug-ins. An Eclipse plug-in is the smallest unit of the platform that implements a well-defined feature.

The Eclipse IDE is a generic IDE built on top of the Eclipse Platform. It has the standard IDE user-interface with multi-window editors, refactoring, syntax highlighting, and other features commonly found in advanced IDEs. It can be used to develop Eclipse-based rich client applications and software written in various programming languages such as C/C++, PHP, Fortran, Python, and Perl. The IDE can be augmented with new features due to its extensible plug-in system. There are more than 1000 Eclipse plug-ins providing rich and diverse features for software development [19]. A number of notable Eclipse plug-ins provide support for parallel programming, data modeling, embedded devices, database development, web services, and so forth.

Eclipse provides a complete suite of frameworks, tools, and environment for application development. Similar to NetBeans, it allows developers to create a specialized tool to suit a particular programming task.

4. INTEGRATED RELATIVE DEBUGGING

As discussed, IDEs offer significant benefits that aid software development. Importantly, many new tools are available in commercial IDEs, and these plug-and-play together creating a very powerful environment. Our experience in integrating Guard into a number of IDEs has highlighted specific advantages to a relative debugger, namely an improved user interaction, interactive assertion management, integrated data visualization, and language neutrality. Here we explore each of these in more detail.

4.1. Improved user interaction

In the command line version of Guard, a user creates assertions by typing information about the structures, files, and line numbers. Thus, in order to create an assertion, a user must have access to the two source programs in order to capture the variable names and line numbers. This is potentially error prone and cumbersome, especially when a large number of assertion are involved.

One of the more significant issues is that assertions are tied to particular line numbers; if the line numbers change, then the assertions are no longer valid. Further, while it is possible to save assertions to a file for later restoration, it becomes unlikely that they will still be correct once the files have been edited and changed.

An IDE offers advantages for both the creation and management of assertions. Since the entire experience is interactive, it is possible to create a point-and-click technique for making assertions in the first place. Most IDEs support multiple concurrent open files that can be viewed and manipulated by language specific source code editors. Many IDEs also allow two totally separate programs to

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

be open at the same time, making it very easy to build an assertion from two different programs. For example, as will be demonstrated later in the paper, a natural interactive interface involves the user pointing to variables in both source windows for both programs, and then requesting the environment extract the variable names and line numbers for the assertion.

It is also much easier to keep assertion line numbers in step with the source code in an IDE. When a source file is edited, the IDE can generate an exception that causes the assertions to be updated. Thus, providing the files are not edited outside the IDE, the assertions can be adjusted automatically. Further, it becomes fairly easy and natural to allow assertions to be saved during an interactive session and restored at a later time. This means that assertions can be used for regression testing as an application evolves.

Integration of the assertion mechanism in the IDE facilitates interaction with other tools' plug-ins. One of the more interesting combinations involves the use of source code control systems, such as Microsoft's SourceSafe or open source projects like CVS and SVN. Source code control systems allow a user to maintain multiple versions of a program, and make it easy to move between versions. When integrated into the IDE, a user can check out a particular version and build it into an executable with ease. It is possible to store partial assertion information, in particular the variable name and line number, as a named *pragma* in a source file. Then when two different versions of a source file are checked into the IDE, these partial assertions can be matched and merged into a set of complete assertions. This technique makes it possible to decide points of interest in a program as it is being written. These test points can then be used later when a version of a program misbehaves, and it can be regression tested to an earlier version.

4.2. Building assertions interactively

The relative debugging methodology, presented in Section 2, defines the steps that a user should follow in order to localize faults. In general, if the output of a computation is reported to be incorrect, the user should define additional assertions to follow the inputs of the faulty computation. To do this, the user will typically locate the definition points of the variables used in the computation. This process usually continues, in an iterative fashion, until the faulty section of code is located and traditional debugging techniques can be used to correct the error.

To locate variable definitions and construct useful assertions require the user to have a detailed knowledge of the two programs under consideration. Without this, the task of locating definitions becomes difficult and time consuming. This is especially true in complex multi-file programs where there may be multiple definitions that are sparsely scattered throughout the program.

By applying the relative debugging methodology, it can be seen that the data flow properties of a program are needed to identify useful data structures and comparison points. Using the data flow properties of programs, it is possible to build a special data flow browser, which locates and displays the definitions that may assign a value to a selected variable in a particular computation. Such a tool provides invaluable support for relative debugging by reducing the complexity of identifying variable definitions. Furthermore, it decreases the need for users to have a detailed knowledge about the program under consideration. Importantly, such a browser can be added as a plug-in to an IDE and can be integrated with the assertion generator. Thus, as a user navigates the Define-Use chains they can build assertions relatively easily.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

4.3. Integrated data visualization

In the command line version of Guard, the results of comparisons are reported to standard out. In an IDE, at the very least, these differences can be reported in a special output window. However, it is possible to improve error reporting by integrating more completely in the environment. For example, many IDEs provide sophisticated tree viewers for examining the state of structured data. Record or struct viewers allow a user to fold and unfold sub-fields. Multi-dimensional arrays can be folded and only those elements of interest can be displayed.

Further, it is possible to interact more closely with third party visualization packages. While this function was supported in the command line version of Guard, it usually involved writing files from the debugger, which could then be displayed in a visualization package like IBM's DX. This technique is extremely powerful and has allowed us to locate very complex errors in the past. However, because the tools are separate, the process tends to be done sequentially—first run the computations, then visualize the results. However, by using a visualization package that is integrated into the IDE, it is possible to use the graphics and displays when refining the assertions.

4.4. Language neutrality

Many IDEs support more than one language system. Specifically, they often provide syntax-directed editors for specific languages, and the variable displays are performed using the same notation as the base language. For example, multi-dimensional arrays, which often have different syntaxes in different languages, adopt the syntax of their host language syntax when manipulated in the environment.

Some IDEs, such as Visual Studio go as far as to define a language neutral virtual machine. In this case, the environment is able to provide access to the internal state of any programming language supported by the base runtime environment. Visual Studio, through the .NET runtime, currently supports an enormous range of different languages, but the interfaces that are exposed to plug-ins are the same. This makes it very easy to build a plug-in that is independent of the host language.

5. THE GUARD-IDE

5.1. Architecture

The plug-in architecture and platforms for IDEs vary significantly, so we have designed the Guard-IDE to be as generic as possible. Figure 2 shows the architecture and identifies two main parts:

- an IDE package, which resides within the target IDE. This provides an implementation of the GUI for managing assertions, and exposes important debugging interfaces in the IDE that are required by the Guard Interpreter. Importantly, while the interface to this package is the same across different IDEs, each implementation requires different code because of the significant differences in platform;
- the Guard package, which is external to the IDE. This provides the core relative debugging functionality (including a sophisticated data-driven execution engine), as well as necessary

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

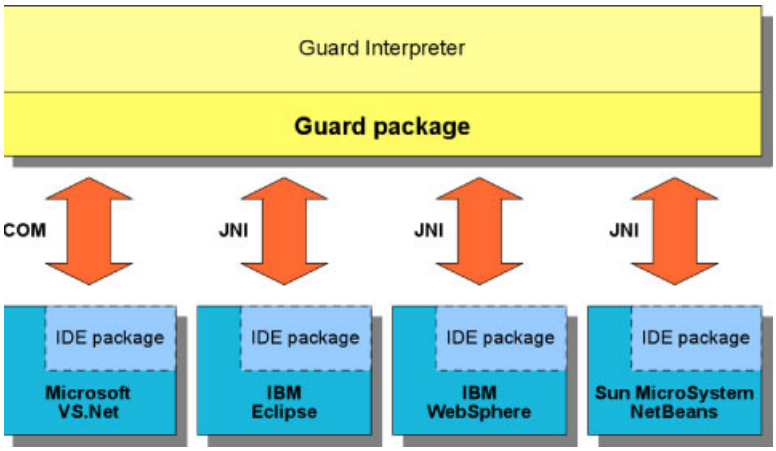


Figure 2. The Guard-IDE architecture.

infrastructure for communication between the IDE and Guard. This component is generic across all implementations.

The Guard package is actually built from original ClassicGuard components. It encapsulates Guard's core relative debugging functionality within a module called the Guard Interpreter. In ClassicGuard, debug functions (such as managing breakpoints, evaluating expressions, etc.) are provided by a separate client–server-based debugger, built on top of GDB. However, in Guard-IDE these debug interfaces are provided instead by the target IDE itself. This leverages the functions that are already available in the IDE, and allows us to support multiple languages in a way that is consistent with the basic IDE operation.

Both the Guard package and the specific IDE packages utilize different communication mechanisms depending on the available infrastructure. The Java-based IDE, such as Eclipse, WebSphere, and Netbeans all use Java Native Interface (JNI) as the communication infrastructure; however, VS.Net uses COM under Windows.

The Guard package is almost identical in each of the different Guard implementations, except for the VS.NET version that uses COM instead of Java. Its interface is shown in Figure 3, and is divided into four main categories. The interface provides functions that:

- start, stop, and control the core debugging engine (Core). A detailed description of the engine, which is based on a dataflow graph interpreter, is beyond the scope of this paper, but is described in more detail in [36];
- build and maintain assertions (Assertion management);
- attach to and manage individual processes (Processor Control); and
- receive notification of breakpoints (Breakpoint).

The first three interface classes are invoked by the IDE in response to user interactions, and the Breakpoint interface is invoked when the IDE is notified of a breakpoint event.

The IDE packages, on the other hand, differ from each other because there is no common standard structure for IDE plug-ins; each system has different techniques and underlying assumptions.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Core:	StartGuard(), StopGuard(), CreateGraph(), StartGraph(), StopGraph()
Assertion management:	AddMap(), AddAssignment(), AddAssertion()
Process Contol:	AttachProc(), InvokeProc(), CreateProcess(), DestroyProcess()
Breakpoint:	BreakpointHit()

Figure 3. Guard package interface.

Process control:	AutoSetLineBreak(), AutoDeleteBreak(), AutoGo(), AutoStop(), AutoBreak()
Data:	AutoEvalExpr()
Display:	DisplayData(), DebugMessage(), ErrorMessage()

Figure 4. IDE package interface.

However, they required roughly the same amount of code to expose the core Guard functions through the user interface. The interface is divided into three classifications, shown in Figure 4, and these provide functions that:

- allow the core debug engine to set and manage breakpoints (Process control). These are required in order to build assertions, and are typically supported by the debugger interface used by the IDE;
- evaluate expressions (Data). This is required to execute assertions when they fire, and again, is supported by the native IDE debugger; and
- display information such as general messages in the IDE (Display).

5.2. Eclipse implementation

While each of the IDEs discussed in Section 3 offer similar user-level functionality, their implementation details differ significantly. As a representative example, we highlight some details of the Eclipse plug-in called EclipseGuard (Figure 5).

EclipseGuard makes significant use of Eclipse's plug-in and extensibility concept. At its core, Eclipse has a modular Java runtime called Equinox. Equinox is an implementation of the OSGi R4 core framework specification, which provides a set of bundles and services required to support running OSGi-based systems. The platform is the middle tier of the architecture, and consists of a set of components that provide core services and frameworks to higher tiers. The top tier of the architecture incorporates plug-in features that provide the functionality most visible to the users.

Importantly, Eclipse plug-ins allow other plug-ins to extend or customize portions of their functionality by declaring *extension points* and *extensions*. For example, many programming languages' development tools are already available in Eclipse, such as JDT for Java development tools and CDT for C/C++ development tools, and these provide language specific editors, views, and debugging

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

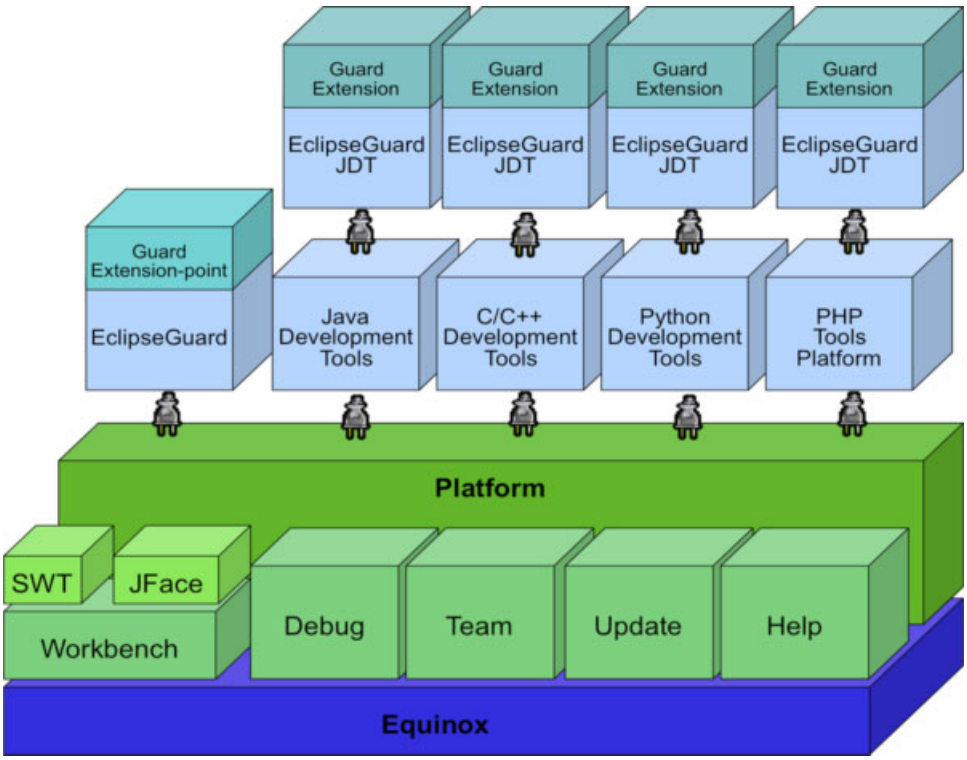


Figure 5. EclipseGuard architecture.

tools, to name a few. Importantly, each of the language plug-ins can be extended by Guard, which allows us to perform relative debugging across multiple languages and development environments.

Figure 6 shows the two main packages of the architecture of EclipseGuard as well as the individual components that define the relationship between Eclipse and Guard.

5.2.1. Guard package

As discussed, the Guard package is external to Eclipse and provides relative debugging functionality by reusing the original components of ClassicGuard. Since the Guard Interpreter is written in C, we implemented an interface layer as a wrapper on top of the Guard Interpreter to interact with the Java-based IDE package. The JNI is used to facilitate calls out of the Guard Interpreter to the IDE package, such as those required to place breakpoints, perform expression evaluation, etc.

5.2.2. IDE package

The Eclipse IDE package is responsible for implementing Guard's GUI and provides functions such as debugger control, setting breakpoints, receiving breakpoint events, and expression evaluation.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

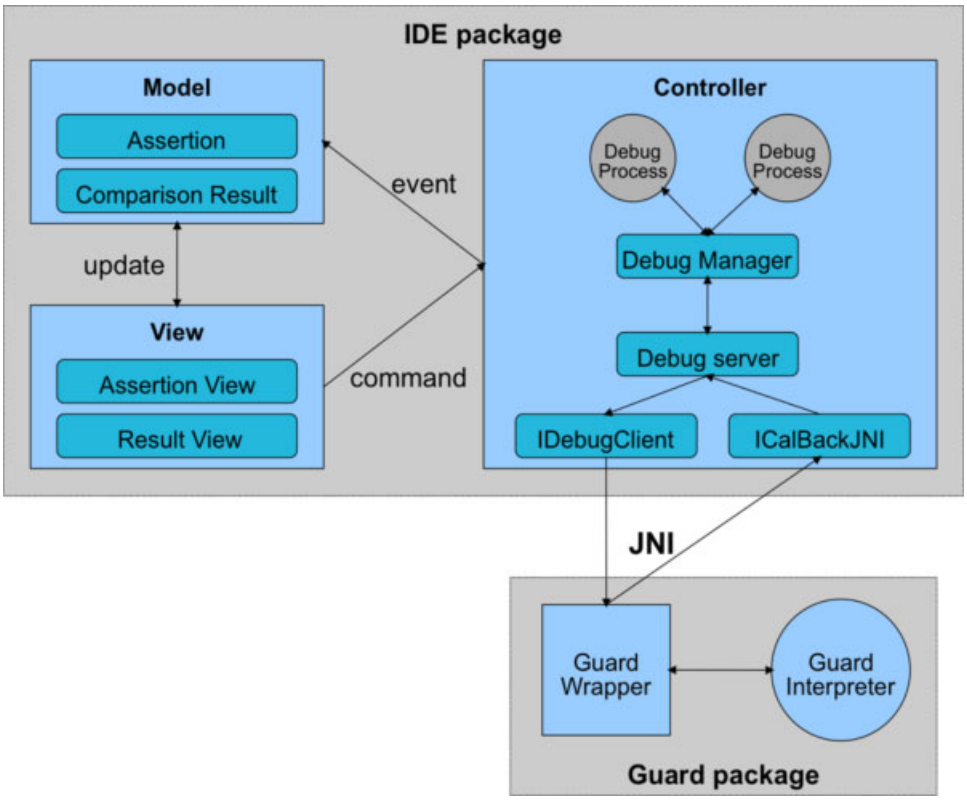


Figure 6. EclipseGuard packages.

This package uses a model-view-controller (MVC) design pattern and provides a generic interface that handles controlling and monitoring of relative debugging:

- A generic high-level relative debug model that represents the assertions and comparison results.
- Views that give the user graphical visualization of the model's status and provide a user interface to control relative debugging by transferring commands.
- A controller that keeps the model updated by generating events based on the messages from the external Guard package.

6. IMPLEMENTATION

In this section we illustrate the ideas discussed in Section 4 through a set of small applications. We deliberately chose small examples to highlight the features of the user interface rather than to show that relative debugging scales to quite large applications (as discussed in Section 2). Importantly, not all of the ideas from Section 4 have been implemented in all IDEs. The table below summarizes which features have been implemented in the various IDEs.

Copyright © 2009 John Wiley & Sons, Ltd.

Feature	Visual Studio	Eclipse	Netbeans
Improved User Interaction	√	√	√
Building assertions interactively	√		
Integrated Data Visualization		√	
Language Neutrality	√	√	√

6.1. Improved user interaction

A screenshot of the EclipseGuard plug-in running under the Eclipse IDE is shown in Figure 7. This screen clearly shows the two source editors, one for Java and the other for C, supported by JDT and CDT plug-ins, respectively. In order to retrieve variable information from different language editors, small third party EclipseGuard language plug-ins are implemented to extend those development tools' plug-ins. The screen also shows information about an assertion in the pane labeled with A, the comparison output for that assertion in pane labeled B, the debug messages in the pane labeled C, and a button to add an assertion (D).

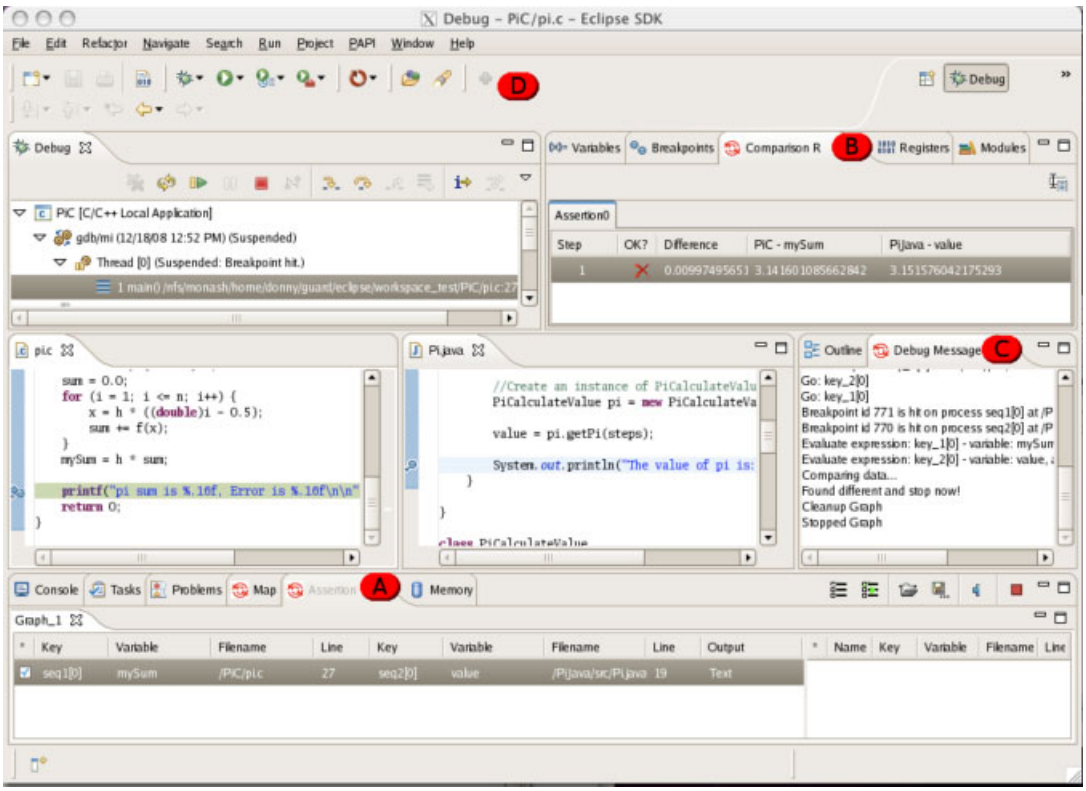


Figure 7. Main screen of EclipseGuard plug-in.

Copyright © 2009 John Wiley & Sons, Ltd.

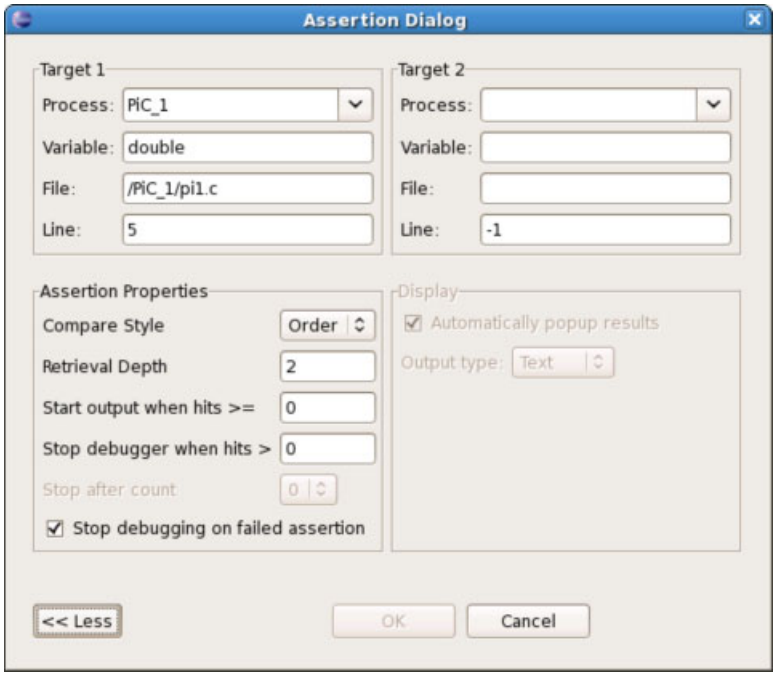


Figure 8. Assertion dialog. Creating an assertion interactively.

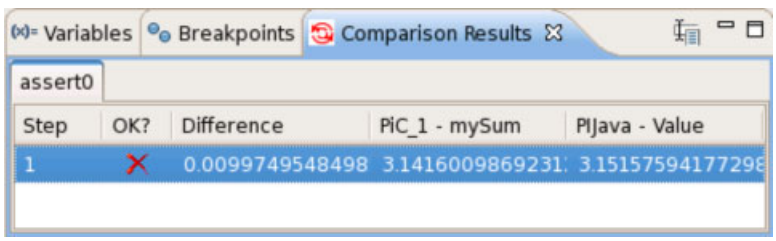


Figure 9. The output of assertion results.

An assertion is created in a few simple steps by selecting a statement in one program and (right) clicking on the variable in a source editor. The variable information such as line number, variable name, source file, and project name are extracted automatically and added to an assertion dialog as shown in Figure 8. The information for the other side of the assertion is added in the same way, except using the alternate source window. The user is also able to specify properties about the assertion such as what comparison method should be displayed, when the debugger should be stopped, and whether the debugger should be stopped on a failed assertion. A completed assertion and its details will be displayed on the Assertion view while the view is able to contain a number of assertions by repeating this process. Thus, it is very easy to create a number of assertions. Importantly, because the system is integrated into the IDE, when the source code is edited and the line numbers change, the assertions are adjusted automatically.

Copyright © 2009 John Wiley & Sons, Ltd.

Once both halves of an assertion have been encountered, the comparison result of the assertion is displayed. After Guard performs the comparison, the result is sent to the IDE package via JNI. The comparison result view shows the assertion result by applying the MVC design pattern. The controller updates the assertion result model and generates an event to notify the viewer to display this model. Figure 9 shows the output of the result of an assertion, in which two scalars have different values.

6.2. Building assertions interactively

To support the ideas discussed in Section 4.2, we have developed a relative debugging data flow browser, named DUCT [37,38], which finds and highlights the definitions of a selected variable at a specified program location. DUCT generates use–define (UD) chains for programs that execute within the Microsoft .NET framework and displays the results within the Microsoft Visual Studio environment. A .NET program is contained within a portable executable (PE) file that contains compiler-generated Microsoft Intermediate Language (MSIL) [39] and metadata [40] that conform to the common type specification (CTS) [41]. MSIL is a CPU-independent instruction set that can be efficiently transformed into native code (by a just-in-time (JIT) compiler) at runtime. Metadata allows a .NET program to be self-describing by detailing the types, signatures, and other data that the .NET framework uses.

In practice, this allows DUCT to be used with a larger number of development languages (i.e. any language that targets the Microsoft .NET framework). For research purposes, the framework provides a common, well-defined platform in which the tool and exercised techniques can be evaluated. To date, DUCT has been successfully tested on .NET programs built by the GPCP [42] compiler as well as the Microsoft compilers for VB, C#, C++, J#.

DUCT has been integrated into Microsoft Visual Studio .NET using the Visual Studio Integration Program (VSIP). VSIP is a component framework that allows Visual Studio .NET to be extended. The framework allows DUCT to be hosted by Microsoft Visual Studio .NET and utilize the IDE and services offered by the environment.

VSIP allows Microsoft Visual Studio .NET to be extended by providing custom functionality in VSPackages. VSPackages are components that implement interfaces defined by VSIP, allowing Microsoft Visual Studio .NET and VSPackages to interact. VSPackages can proffer services that other VSPackages consume. Microsoft Visual Studio .NET also exposes a number of VSPackages that proffer a broad range of services and provide access to the environment's windowing functionality.

To locate the definition(s) reaching a variable, the user must select the variable in the source program and inform DUCT to locate the definitions. This is achieved by:

- Placing the cursor over the variable of interest.
- Selecting 'Construct UD Chain' on the popup context menu, which is activated by clicking the right mouse button.

DUCT currently recognizes scalar variables, objects, object member fields, and arrays. The located definition(s) are clearly highlighted in the source code window of the IDE and listed in a separate window, allowing the user to easily navigate through the definitions.

For example, Figure 10 illustrates the UD chain constructed by DUCT for the variable x in line 22 in a simple C# program. Each definition that reaches the use of x in line 22 is highlighted in

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

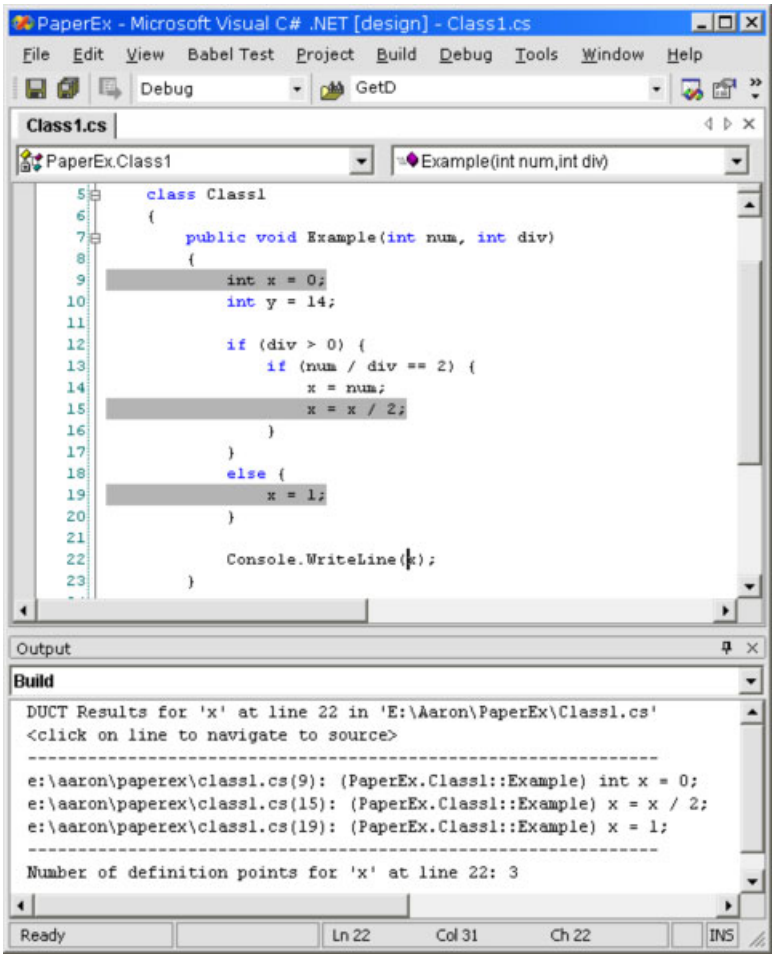


Figure 10. Example DUCT screenshot.

the source code window and detailed in the output window at the bottom of the IDE. The user may navigate to a particular definition by clicking on the definition in the output window. This is particularly useful when a definition is not displayed on the screen or is located in another file.

In our experience, the use of a tool like DUCT improves the user productivity significantly because the navigation of UD chains becomes much simpler—especially when the code is spread over many screens and even source files.

6.3. Integrated data visualization

As discussed, it is possible to leverage the existing display techniques when a relative debugger is integrated into a particular IDE. Here we show that EclipseGuard is able to reuse the standard display methods to not only show state values, but also differences in values.

Copyright © 2009 John Wiley & Sons, Ltd.

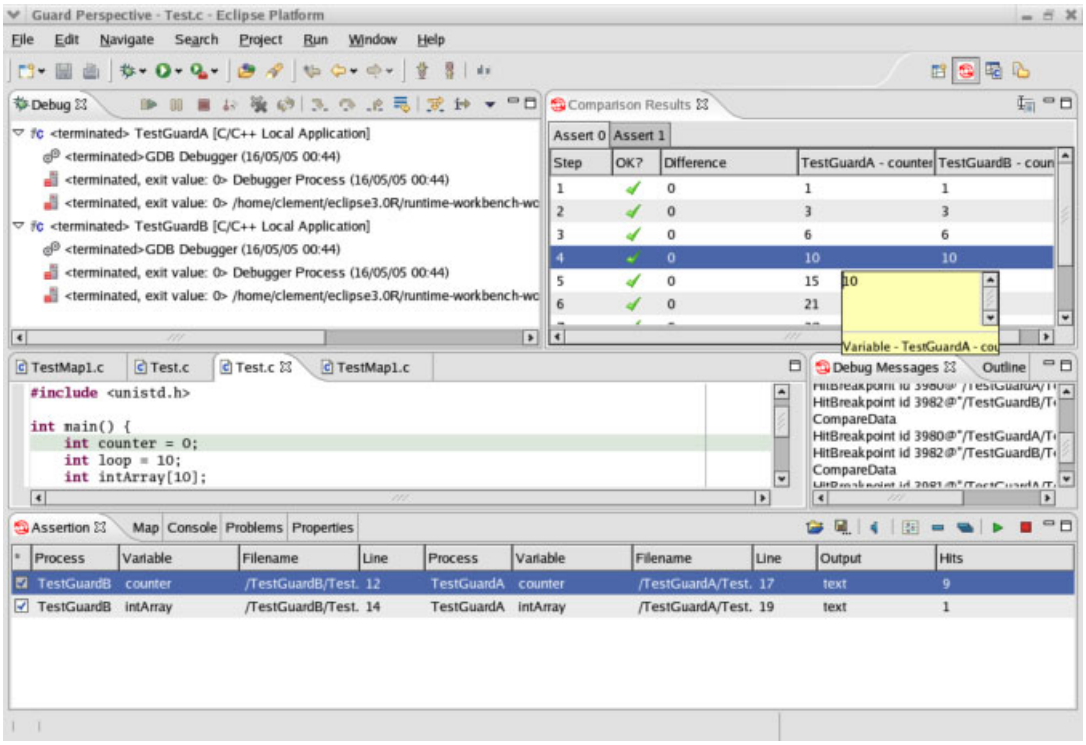


Figure 11. EclipseGuard comparison results.

Figure 11 shows differences reported between two simple scalars being evaluated within a loop. Consequently there are multiple values reported for a single variable. Here it is only necessary to report the values of the two variables at each iteration, the difference (in this case 0) and a graphical icon that shows whether they are the same or not (in this case, ticks).

Figure 12, on the other hand, shows the display of differences in dynamic structured data types. Here, EclipseGuard has traversed a linked list, displayed each of these, and has also displayed the difference between the structures as a list. Again, it uses a simple graphical icon to indicate that the structures are actually the same.

Figure 13 shows how a tree viewer can help display much more complex structures. In this case, the display shows the values from either structure and clearly highlights (using a shaded line) where the structures differ—in this case, the data was not only different but the type definitions were incompatible.

Finally, Figure 14 shows the difference between two arrays, with the locations of the differences highlighted. This particular visualization can display multi-dimensional arrays by taking two-dimensional slices.

An important aspect of all of these data visualization techniques is that they leverage interfaces that are already available in Eclipse or the various plug-ins. For example, the Array Viewer shown

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

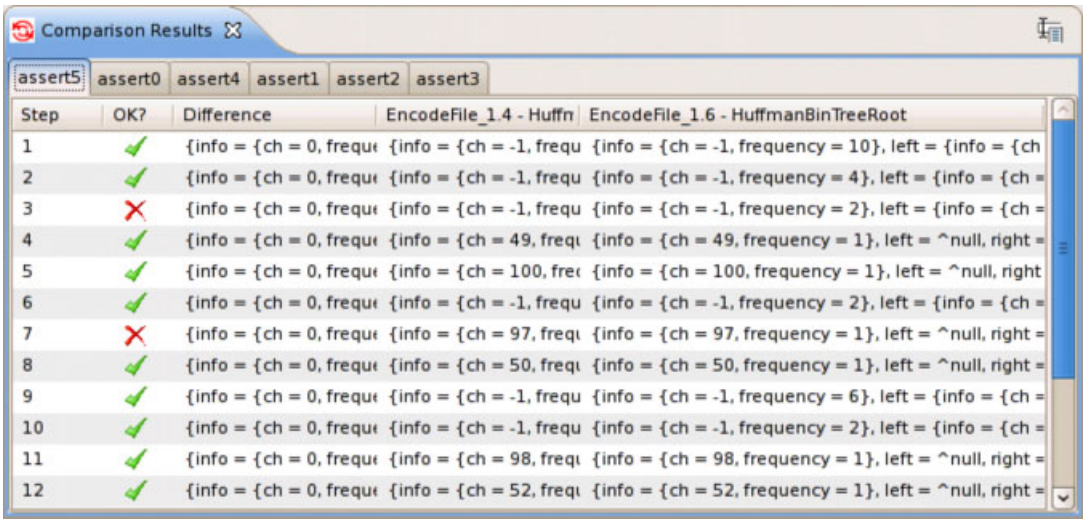


Figure 12. Display of structured data.

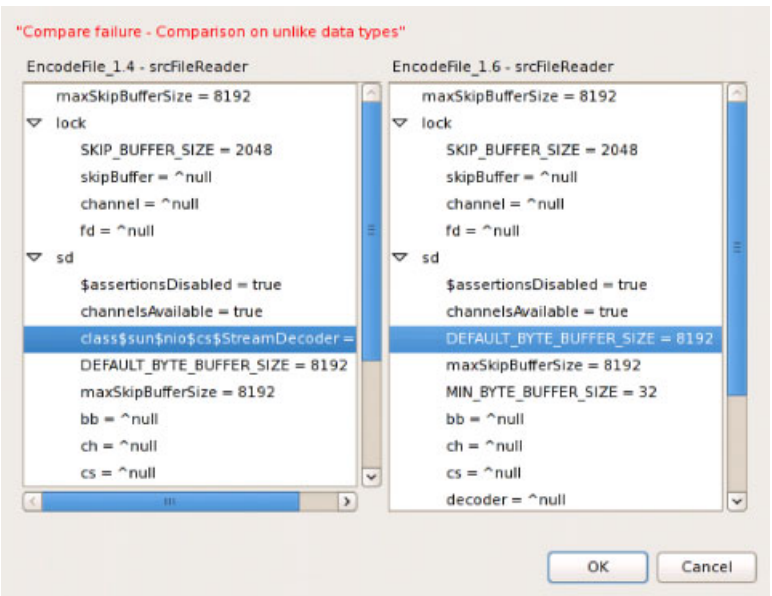


Figure 13. Leveraging native structure viewers.

in Figure 12 was originally developed as part of the Parallel Tools Platform (PTP) project [43], but is equally accessible in Eclipse Guard.

On the other hand, it is also possible to invoke powerful external packages to augment the internal mechanisms. For example, Figure 15 shows the types of images that are possible using external

Copyright © 2009 John Wiley & Sons, Ltd.

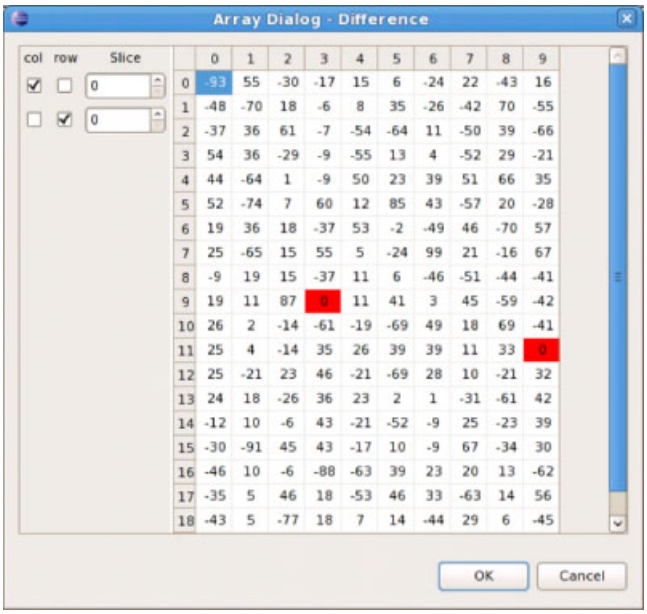


Figure 14. Array viewer.

visualization packages. In this example, a scientific code simulates the effect of dropping a solid object into a tank of liquid. The system is modeled as a collection of discrete, but interacting, particles, governed by Newton's laws of motion. In Figure 15 we show the contents of the key data structure for both the reference (a) and debugged code (b) where each particle is mapped to its coordinates in two dimensions, as produced by IBM Data Explorer. While each of these visualizations is useful in its own right, it is very difficult to see any difference between them when shown as separate images. However, when Guard subtracts the two images, the differences become visible. By augmenting each particle to indicate the magnitude of the difference in position (size of each particle) and speed (grey scale density or colour in online version), it is clear that most particles have different positions, and some have dramatically different speeds.

6.4. Language neutrality

All three IDEs exhibit some degree of language neutrality and allow users to manage source projects in different languages. However, Visual Studio not only maintains a common environment across a huge range of languages, but also leverages the .NET machinery to provide a language neutral execution (and thus debugging) platform. VSGuard makes significant use of this functionality and allows cross language relative debugging.

In one case study, we used VSGuard to find errors in a program that was converted from Visual Basic to C# [38]. The program used in this case study accepts a number expressed in base 2, 8, 10, or 16, via a graphical interface, and converts it to the equivalent number in base 2, 8, 10, or 16 as requested by the user. The first problem we encountered was that the ported program failed to

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

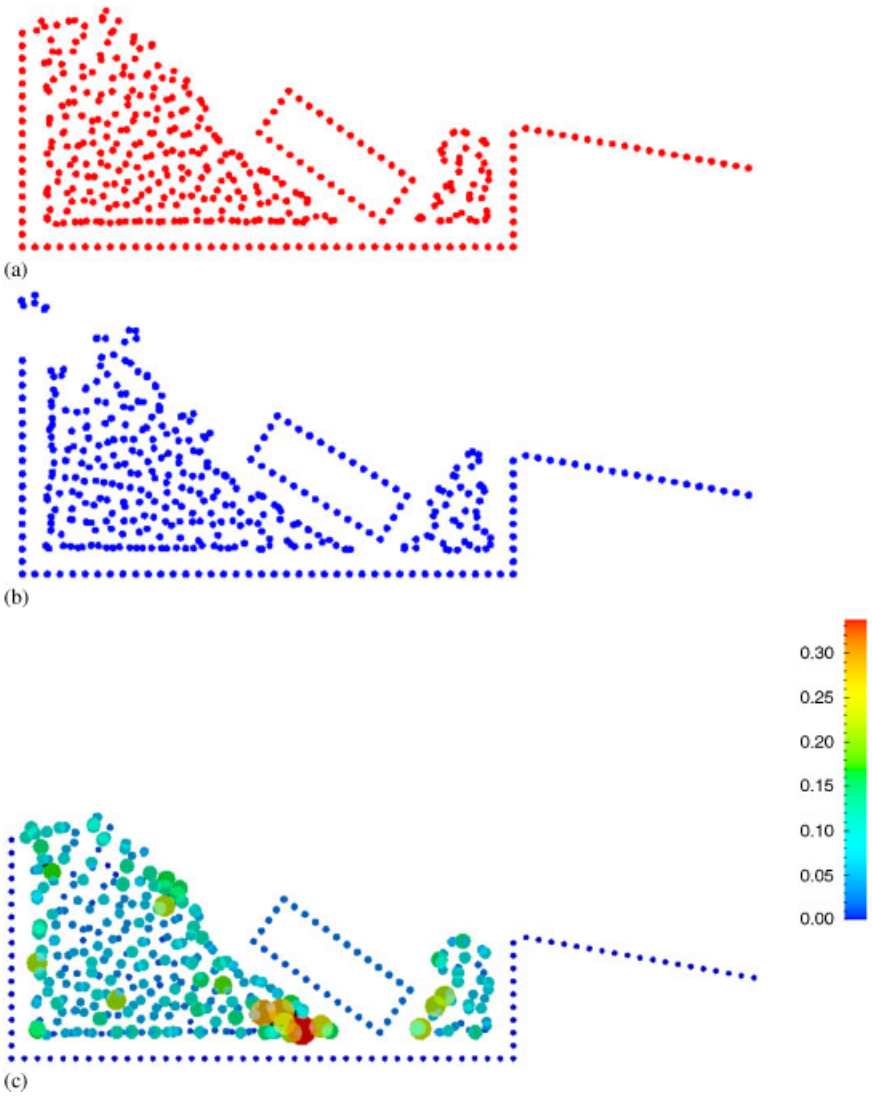


Figure 15. Scientific visualization of differences: (a) and (b) are original and (c) is difference.

display a result when converting a number from base 16 to base 10. To determine the source of the problem, we located the functions, in the original and ported program, that perform the conversion. Figure 16 shows the original VB .NET code and the ported C# code in the left and right source code panes, respectively. Using VSGuard, it was possible to build assertions across the two programs, execute them, and trace the error. Because the runtime system and Visual Studio present uniform interfaces for both languages, this debugging session proceeded without regard to the language.

Copyright © 2009 John Wiley & Sons, Ltd.

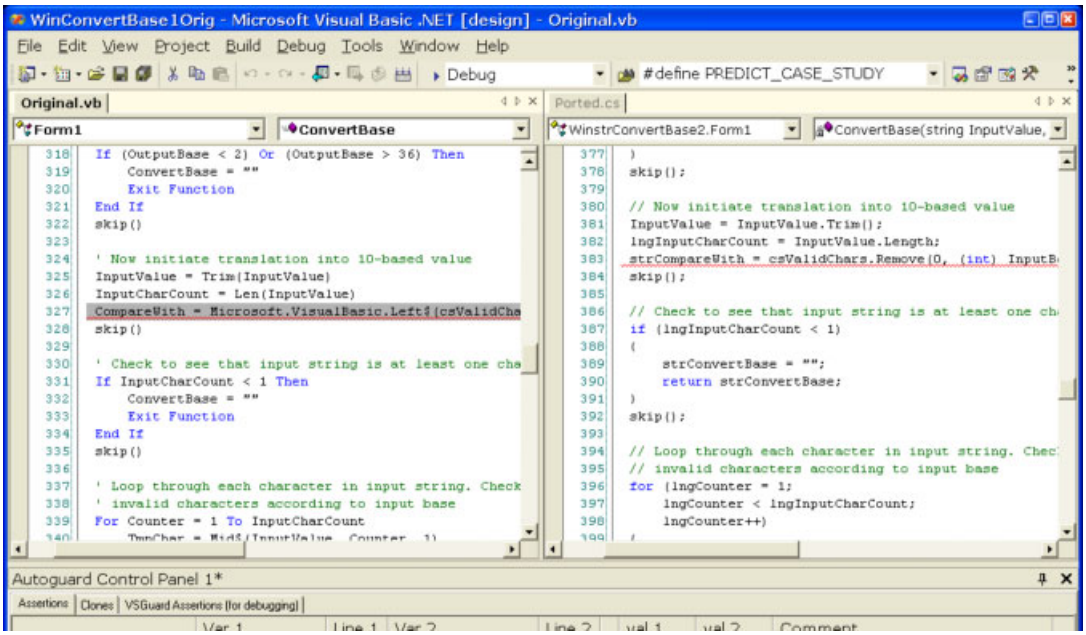


Figure 16. Multiple languages in VSGuard.

7. FUTURE WORK AND CONCLUSIONS

In this paper we have shown how our prior work in relative debugging can be extended to operate in IDEs, and importantly, we have shown that there are significant advantages in doing this. Moreover, we were able to build additional tools, like the DUCT dataflow browser, and use this to support the creation of assertions—one of the more difficult activities in relative debugging. The rich environment of an IDE also brings powerful data visualization tools, and these can be adapted to work with a relative debugger, further enhancing the system capabilities. Finally, the focus on a single environment across different languages means that we can debug two programs that are supposed to perform the same function, even when they are written in different languages.

Some of the capabilities not discussed in this paper concern debugging parallel programs. We are currently involved in two projects that should deliver significant results in this area. First, the Eclipse PTP [43] concerns the development of an open and extensible parallel debugger in an IDE. We have contributed a significant number of components to PTP to date. Second, we are currently converting EclipseGuard so that it leverages the PTP runtime, making it possible to use relative debugging on parallel applications. Even though much of this capability was previously demonstrated in ClassicGuard [3], we have not been able to show this functionality in an IDE.

We are currently investigating extensions to EclipseGuard to allow it to better accommodate complex project structures. For example, at present, we assume that the two programs being compared are actually contained in two separate projects. However, in real world applications of Eclipse, users often build programs across multiple projects, which means we need a more complex project management approach in EclipseGuard. We are also exploring more powerful data

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

visualization techniques and integration with source code control systems as discussed in Section 3.2.2. We also have an experimental version of an automatic debugging tool based on relative debugging, called AutoGuard. AutoGuard builds on DUCT, and is able to automatically generate assertions between programs under some circumstances. This has the potential to dramatically improve user efficiency and interaction [38].

ACKNOWLEDGEMENTS

This work has been supported by the Australian Government through the ARC Discovery grant scheme. It has also been supported by Object Technology International, Microsoft, IBM, and Sun. We wish to thank various colleagues who have contributed over the years, including Wojtek Goscinski, Tim Ho, Le Phu, and Greg Watson.

REFERENCES

1. Abramson D, Foster I, Michalakes J, Sosic R. Relative debugging: A new paradigm for debugging scientific applications. *The Communications of the Association for Computing Machinery* (*CACM*) 1996; **39**(11):67–77.
2. Abramson D, Foster I, Michalakes J, Sosic R. Relative debugging and its application to the development of large numerical models. *Proceedings of IEEE Supercomputing 1995*, San Diego, CA, U.S.A., December 1995.
3. Watson G, Abramson D. Relative debugging for data parallel programs: A ZPL case study. *IEEE Concurrency* 2000; **8**(4):42–52.
4. Available at: http://www.guardsoft.com [16 June 2009].
5. Available at: http://www.guardsoft.com/news.html [16 June 2009].
6. Agrawal H. Towards automatic debugging of computer programs. *PhD Thesis*, Software Engineering Research Centre (SERC), Purdue University, 1991; 194.
7. Ducassé M, Emde A. A review of automated debugging systems: Knowledge, strategies, and techniques. *Tenth International Conference on Software Engineering* (*ICSE 10*), Singapore, 1988; 162–171.
8. Shapiro E. *Algorithmic Program Debugging*. MIT: Cambridge, U.K., 1982.
9. Whalley DB. Automatic isolation of compiler errors. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*) 1994; **16**(5):1648–1659.
10. Reps T, Ball T, Das M, Larus J. The use of program profiling for software maintenance with applications to the year 2000 problem. *Sixth European Software Engineering Conference* (*ESEC 97*) *held jointly with the Fifth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE 97*), Zurich, Switzerland, 1997; 432–449.
11. Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 2002; **28**(2):183–200.
12. Renieris M, Reiss S. Fault localization with nearest neighbor queries. *Eighteenth IEEE International Conference on Automated Software Engineering*, Montreal, Que., Canada, 2003; 30–39.
13. Zeller A. *Why Programs Fail*: *A Guide to Systematic Debugging*. Morgan Kaufmann: Los Altos, CA, October 2005. ISBN: 1558608664.
14. Brown A, McDermid J. Learning from IPSE's mistakes. *IEEE Software* 1992; **9**(2):23–28.
15. Kurtz T. BASIC. *History of Programming Languages*, Wexelblat R (ed.). Academic Press: New York, 1987; 515–537.
16. Boekhoudt C. The big bang theory of IDEs. *ACM Queue* 2003; **1**(7):74–82.
17. Fuggetta A. A classification of CASE technology. *IEEE Computer* 1993; **26**(12):25–38.
18. Sommerville I. *Software Engineering* (8th edn). Addison-Wesley: Reading, MA, 2006.
19. Kline R, Seffah A. Evaluation of integrated software development environments: Challenges and results from three empirical studies. *International Journal of Human–Computer Studies* 2005; **63**(6):607–627.
20. Reis C, Cartwright R. Taming a professional IDE for the classroom. *SIGCSE 2004*: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, VA, U.S.A., 3–7 March 2004; 156–160.
21. Case A. Computer-aided software engineering (CASE): Technology for improving software development productivity. *ACM SIGMIS Database* 1985; **17**(1):35–43.
22. Glass R. The realities of software technology payoffs. *Communications of the ACM* 1999; **42**(2):74–79.
23. Norman R, Nunamaker J. CASE productivity: Perceptions of software engineering professionals. *Communications of the ACM* 1989; **32**(9):1102–1108.
24. Tsuda M, Morioka Y, Takadachi M, Takahashi M. Productivity analysis of software development with an integrated CASE tool. *ICSE 1992*: *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, 11–15 May 1992; 49–58.

Copyright © 2009 John Wiley & Sons, Ltd.

25. Zeller A. The future of programming environments: Integration, synergy, and assistance. *FOSE 2007*: *Future of Software Engineering 2007*. IEEE Computer Society: Washington, DC, U.S.A., May 2007; 316–325.
26. Available at: http://msdn.microsoft.com/vstudio/ [22 January 2008].
27. Available at: http://msdn.microsoft.com/vsip/ [22 January 2008].
28. http://www.netbeans.org/products/platform/ [27 August 2007].
29. http://www.netbeans.org/products/ide/ [27 August 2007].
30. http://java.sun.com/products/jfc/tsc/articles/architecture/ [28 August 2007].
31. http://ant.apache.org/ [28 August 2007].
32. http://subversion.tigris.org/ [28 August 2007].
33. http://www.eclipse.org/ [29 August 2007].
34. des Rivieres J, Beaton W. Eclipse platform technical overview. Available at: http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html [29 August 2007].
35. Yang Z, Jiang M. Using eclipse as a tool-integration platform for software development. *IEEE Software* 2007; **24**(2): 55–67.
36. Watson G. The design and implementation of a parallel relative debugger. *PhD Thesis*, School of Computer Science and Software Engineering, Monash University, Clayton, Australia, 2001.
37. Searle A, Gough JK, Abramson DA. DUCT: An interactive define-use chain navigation tool for relative debugging. *AADebug'03*, Ghent, Belgium, September 2003.
38. Searle A. Automatic relative debugging. *PhD Thesis*, Queensland University of Technology, 2006.
39. Microsoft. Common language infrastructure (CLI): Partition III: CIL instruction set. *ECMA TC39/TG3*, 2001.
40. Microsoft. Common language infrastructure (CLI): Partition II: Metadata definition and semantics. *ECMA TC39/TG3*, 2001.
41. Microsoft. Common language infrastructure (CLI): Partition I: Concepts and architecture. *ECMA TC39/TG3*, 2001.
42. Available at: http://www.plas.fit.qut.edu.au/gpcp/ [16 June 2009].
43. Available at: http://www.eclipse.org/ptp/ [16 June 2009].

Copyright © 2009 John Wiley & Sons, Ltd.

P/00/009
Regulation 3.2

AUSTRALIA

Patents Act 1990

DRAFT

PROVISIONAL SPECIFICATION

Invention Title: **METHOD FOR TESTING AND DEBUGGING COMPUTER PROGRAMS**

The invention is described in the following statement:

THIS INVENTION relates to a method of testing and debugging an executable behaviourally unknown computer program by comparing the behaviourally unknown computer program against an executable behaviourally known computer program. The invention is particularly useful, but not necessarily limited to, testing and debugging in which both programs have been compiled from the same source code or when both programs originate from similar algorithms having identical variable values at specific break points.

Conventional software debugging tools allow a user to interrogate the state and control the execution of a computer program. Such debugging tools require the user to have knowledge of the correct operation of the program in order to analyse various states to thereby find errors. Unfortunately, this may be a problem when the program is poorly structured, sparsely documented or when non-descriptive variable names are used.

Once a computer program (behaviourally known program) has been debugged and operates correctly on one type of computer, having a particular compiler and operating system, there is no guarantee that the program will behave as required either compiled by a different compiler or when run in a different environment.

Consequently, comparison methods have been developed in which the behaviourally known program,

functioning in a desired manner, is compared with a program under test (behaviourally unknown program) having the same source origin as the behaviourally known program. Comparison methods may also be used to test and debug programs originating from similar algorithms having identical variable values at specific break points. The currently available comparison methods are usually effected in an ad-hoc fashion in which either respective program states are dumped to a file and subsequently compared manually or two debugging sessions are required to manually compare respective program states.

If the state is written to a file, then a file comparison program is used to detect the differences. If the output is compared manually, the user must set breakpoints in the two programs, dump the variables manually, and then compare the values.

Further, a hybrid method has also been used in which two debugging sessions dump the values of the variables to a file after which a file comparison program is run to detect differences.

Although more efficient than conventional debugging tools, the above comparison debugging methods are relatively time consuming. Further, they do not directly display differences to a user nor do they not allow the user to interactively view either differences in program states or differences in specified variables.

It is an object of the invention to overcome or alleviate at least some of the problems with prior art comparison debugging methods.

According to one form of the invention there is provided a computerised method for testing and debugging an executable behaviourally unknown computer program the method including the steps of:

(i) controlling the execution of one or more instructions of an executable behaviourally known computer program;

(ii) obtaining a value of at least one variable resulting from step (i);

(iii) controlling the execution of one or more corresponding instructions of said behaviourally unknown computer program;

(iv) obtaining a value of at least one variable resulting from step (iii);

(v) comparing said value of at least one variable resulting from step (ii) with said value of a corresponding at least one variable resulting from step (iv); and

(vi) outputting differences resulting from said comparing.

Alternatively, according to another form of the invention there is provided a computerised method for testing and debugging an executable behaviourally unknown computer program, the method including the steps of:

(a) allowing the selecting of a break point for said behaviourally unknown computer program and an executable behaviourally known computer program;

(b) controlling the execution of said behaviourally known computer program until the break point is reached;

(c) obtaining a value of at least one variable resulting from step (b);

(d) controlling the execution of said behaviourally unknown computer program until said break point is reached;

(e) obtaining a value of at least one variable resulting from step (d);

(f) comparing said value of at least one variable of said behaviourally known computer program with said value of a corresponding variable of said behaviourally unknown computer program; and

(g) outputting differences resulting from said comparing.

Suitably, both said programs may have been compiled from the same source code origin. Alternatively, both said programs may have originated from similar or identical algorithms in which both said algorithms, when functioning in a desired manner, have identical variable values at selected break points.

Preferably, the method may be further characterised by the step of allowing the user the

choice of selecting which variables are to be compared by comparing step (v) or comparing step (f).

Suitably, the method may be further characterised by said programs being executed on different computer means. The method may be suitably effected on a computer means which executes the behaviourally known computer program. Alternatively, said programs may be executed on the same computer means.

Preferably, the method is further characterised by the step of outputting being effected by a visual display unit.

The method may also suitably include the step of interacting, after the step of outputting, wherein the interacting allows the user to control the method such that steps (i) to (vi) or steps (a) to (g) are repeated.

Preferably, the comparing step of (v) may include the steps of:

(vii) subtracting said value of at least one variable resulting from step (ii) with said value of a corresponding at least one variable resulting from step (iv); and

(viii) determining if the modulus resulting from the step (vii) is greater than a threshold value.

Preferably, the comparing step of (f) may include the steps of:

(h) subtracting said value of at least one variable of said behaviourally known computer program with said

value of a corresponding variable of said behaviourally unknown computer program; and

(i) determining if the modulus resulting from the step (h) is greater than a threshold value.

Suitably, the threshold value may be pre-defined.

In order that the invention may be readily understood and put into practical effect, reference will now be made to preferred embodiments illustrated in the accompanying drawings in which:-

FIG 1 is a flow diagram of a first embodiment of the invention;

FIG 2 is a flow diagram of a second embodiment of the invention;

FIG 3 is a flow diagram of the comparing step of FIG 1 and FIG 2;

FIG 4 is a flow diagram of a calculate function used in FIG 3; and

FIG 5 is pseudo-code illustrating steps in the calculate function of FIG 4.

Referring to FIG 1 there is illustrated a comparison method for testing and debugging a computer program in which the user selects an executable behaviourally known computer program or reference program Rp to be executed on a computer Cr shown at step 1. The user also selects an executable behaviourally unknown computer program which is the program to be tested Tp, shown at step 2, in which Tp is to be executed on a computer Ct. Program Tp has

been compiled from the same source code as the reference program Rp. Computers Cr and Ct are linked to communicate with each other and the selection of Rp and Tp includes the identification of their respective paths.

At step 3 at least one program instruction of Rp is then executed on Cr and a corresponding section of program instructions of Tp are executed on Ct at step 4.

By default a single program instruction is executed at each of steps 3 and 4. However, the user can determine the number of instructions to be executed at steps 2 and 3 or alternatively the user can select break points.

Values of variables of Rp and Tp are obtained (read) and then compared at 5 by a comparison program on Cr. These variables may be selected by the user if so desired. If a difference results, the differences are communicated at step 6 to the user via a visual display unit.

Certain errors such as inexact equality in floating point numbers are insignificant and therefore the comparing of step 5 returns a difference only when the result of the comparison is greater than a threshold value. The threshold value is set by the user and has a default value of 0. If the difference displayed at step 6 are insignificant a user may decide to continue debugging, at step 7, by selecting

the Yes option. Alternatively, the user may terminate the debugging, at step 7, by selecting the No option.

Referring back to step 5, if the comparison of Rp and Tp results in the same variable values being identified, steps 3, 4 and 5 are repeated until step 8 results in a Yes which corresponds to there being no more executable program instructions. Accordingly, an error free Tp status is reported at step 9 to the user via the Visual Display Unit.

Referring to FIG 2 a second preferred embodiment is shown in which the user selects the executable behaviourally unknown computer program or reference program Rp on computer Cr at step 10. The executable behaviourally unknown computer program which is the program to be tested Tp on computer Ct is selected at step 11. The paths of both Rp and Tp are also identified at steps 10 and 11.

Break points and variables to be compared are selected at step 12 and Rp is run on Cr at 13 until step 14 detects a break point. Tp is then run on Ct at step 15 until step 16 detects a break point.

The values of the selected variables of Rp and Tp are obtained (read) and then compared at step 17 in which the comparison is executed on Cr. If the comparison results in a difference, the differences are communicated at step 18 via the Visual Display Unit. The user may decide at step 19 to continue selecting break points and variables by selecting the

Yes option. Alternatively, the testing and debugging session can be terminated by selecting the No option.

If at step 17 the comparison of the variables is equal and the end test of Rp and Tp at step 20 results in a NO the method returns to step 12. Alternatively, if at step 20 the result is YES then the end of both programs Rp and Tp have been reached and step 21 reports to the user, via the Visual Display Unit, that there are no errors in Tp and the method terminates.

Referring to FIG 3 there is illustrated the comparing step (5 or 17) applicable both FIGS 1 and 2.

At step 22 respective variables A and B are selected from Rp and Tp. Variable A is then tested at step 23 to determine if it is of type dynamic data structure such as a pointer. If variable A is a dynamic data structure then at step 24 a graph of the data structures of both variables A and B is constructed. At step 25 each node in both graphs is compared by the calculate function of FIG 4 in which each node is compared in a depth first fashion.

If at step 23 it is determined that variable A is not a dynamic data structure step 26 determines whether or not variable A is an array or string. If variable is an array or string then at step 27 each element thereof is compared by the calculate function of FIG 4.

If at step 26 it is determined that variable A is not an array or string then it is assumed that A is an

integer, real, boolean or character. Accordingly, the calculate function of FIG 4 is executed once to compare A and B.

If at step 29 it is determined that there are more variables to be tested the comparing method returns to step 22, otherwise the method terminates.

Referring to FIG 4, step 30 determines if variable A is an integer or real number. If variable A is an integer or real number then at step 31 B is subtracted from A and the modulus of the result is compared against a threshold value. This value can be pre-defined if so desired. If the result is less than the threshold value then A and B are considered equal and an equal condition is returned at step 32. If the result is greater than the threshold value a difference condition is returned at step 33.

At step 30, if variable A is neither an integer or real then step 34 determines if A is a boolean or character. If A is neither a boolean nor a character an error condition is detected at 35 which terminates the method. If A is either a boolean or character then step 36 determines if A is the same as B and an equal condition is returned at step 32. If A is different from B then a difference condition is returned at step 33. After steps 32 or 33 the method terminates.

The threshold value provides a means of reducing or eliminating errors in inexact equality. This

threshold value can be either pre-determined by either a user or it may have an appropriate default value.

In addition to FIGS 3 and 4 reference is now made to the pseudo-code of FIG 5. The function simple_types_equal illustrates how FIG 4 is implemented. Note the subtraction operator and less than operator have been overloaded so that it can test integers, booleans, reals and characters.

The function one_dim_arrays_equal tests one dimensional arrays or strings. Because it is possible that arrays may be of different sizes (due to compilation differences or otherwise), the maximum of the lower boundary and the minimum of the higher boundary are considered.

The function multi_dim_arrays_equal is similar to that of the function one_dim_arrays_equal. As implemented, and as shown, a different function is required for each array dimension. However, a recursive function or a function in which the dimension of the array is passed to the function may be used.

The above embodiments describe two methods for testing and debugging by comparing two programs running on different computers. The method as illustrated in FIGS 1 and 2 are suitably for testing and debugging executable programs compiled from the same source code. The method as illustrated in FIG 2 is also suitably for testing and debugging programs

originating from identical of similar algorithms in which specific variables will be identical at selected break points. The methods as described are also suitable for testing and debugging programs running on the same computer, in such a case Cr and Ct are the same computer.

Although, the flow diagrams show the computers Cr and Ct running sequentially, they can also run concurrently. Further, both the embodiments may have a counter or timer for handling error conditions such as when one of the programs do not return from a break point or if an infinite loop is entered.

The invention has been described with reference to preferred embodiments, however, it is to be understood that the invention is not limited to the specific embodiments hereindescribed.

DATED this Fourteenth day of April 1994

GRIFFITH UNIVERSITY

By their Patent Attorneys

FISHER & KELLY

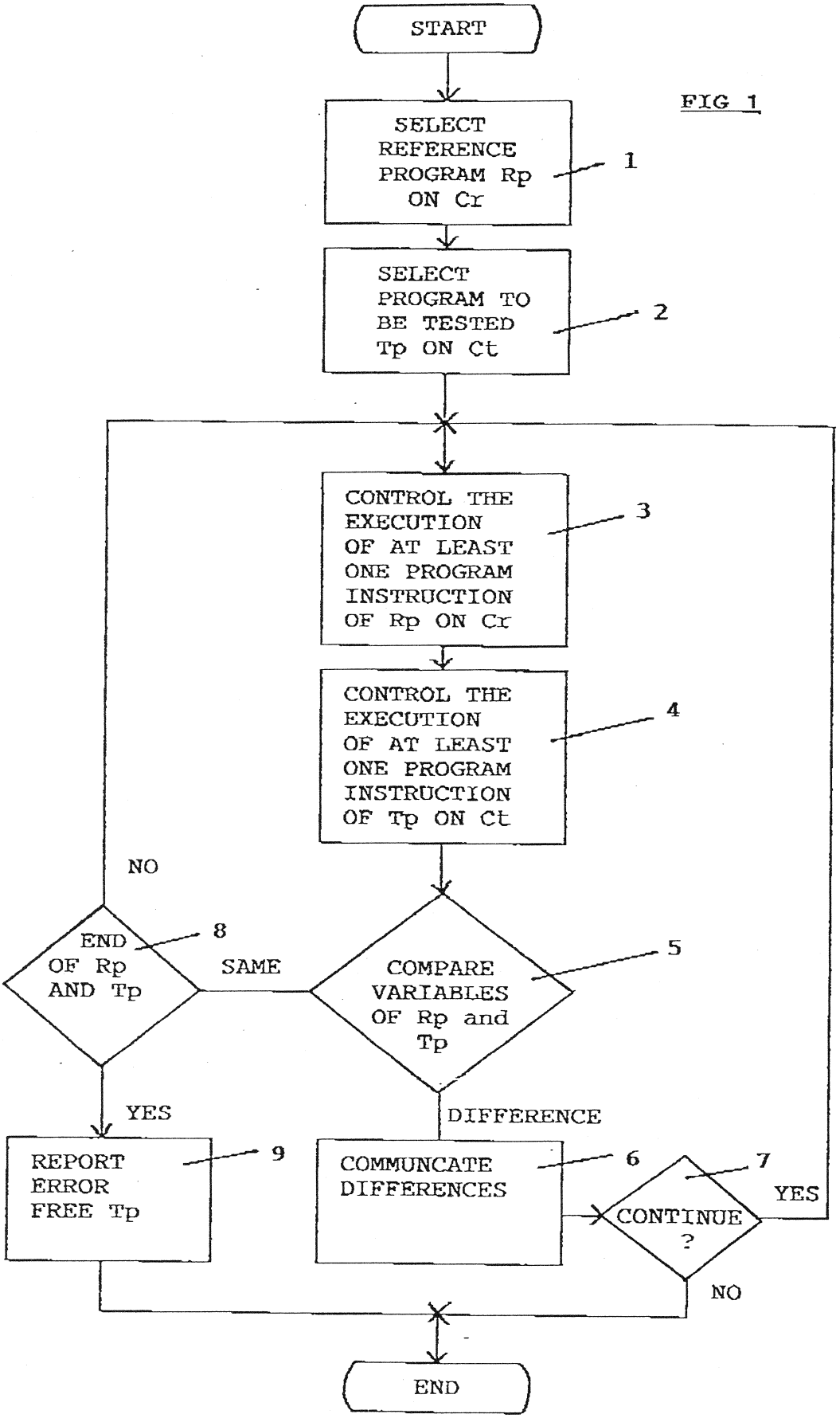

START
SELECT REFERENCE PROGRAM Rp ON Cr
1
SELECT PROGRAM TO BE TESTED Tp ON Ct
2
CONTROL THE EXECUTION OF AT LEAST ONE PROGRAM INSTRUCTION OF Rp ON Cr
3
CONTROL THE EXECUTION OF AT LEAST ONE PROGRAM INSTRUCTION OF Tp ON Ct
4
NO
END OF Rp AND Tp
8
SAME
COMPARE VARIABLES OF Rp and Tp
5
YES
DIFFERENCE
REPORT ERROR FREE Tp
9
COMMUNCATE DIFFERENCES
6
CONTINUE ?
7
YES
NO
END

FIG 1

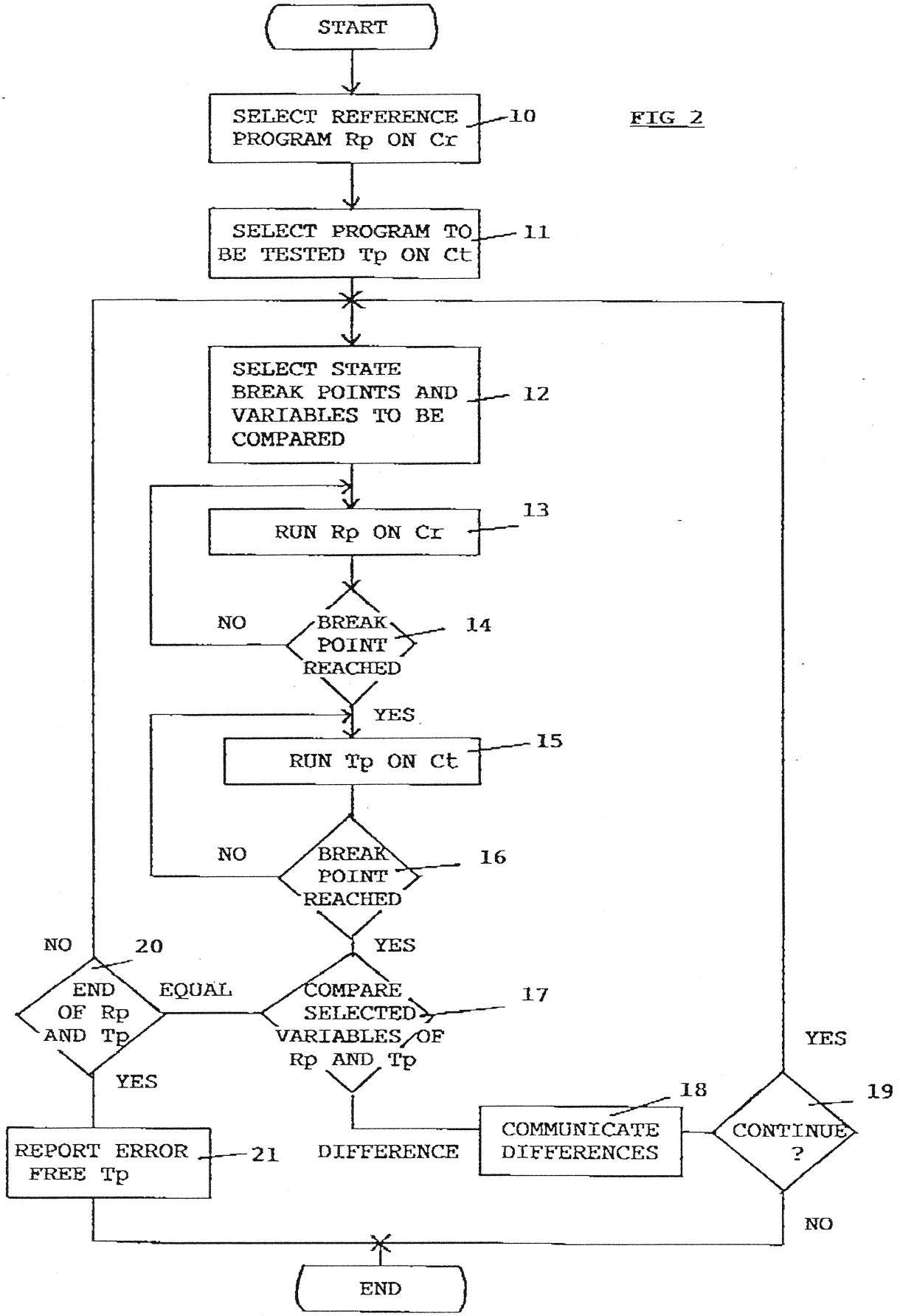

START
SELECT REFERENCE PROGRAM Rp ON Cr
10
SELECT PROGRAM TO BE TESTED Tp ON Ct
11
SELECT STATE BREAK POINTS AND VARIABLES TO BE COMPARED
12
RUN Rp ON Cr
13
NO
BREAK POINT REACHED
14
YES
RUN Tp ON Ct
15
NO
BREAK POINT REACHED
16
YES
NO
20
END OF Rp AND Tp
EQUAL
COMPARE SELECTED VARIABLES OF Rp AND Tp
17
YES
REPORT ERROR FREE Tp
21
DIFFERENCE
COMMUNICATE DIFFERENCES
18
CONTINUE ?
19
YES
NO
END

FIG 2

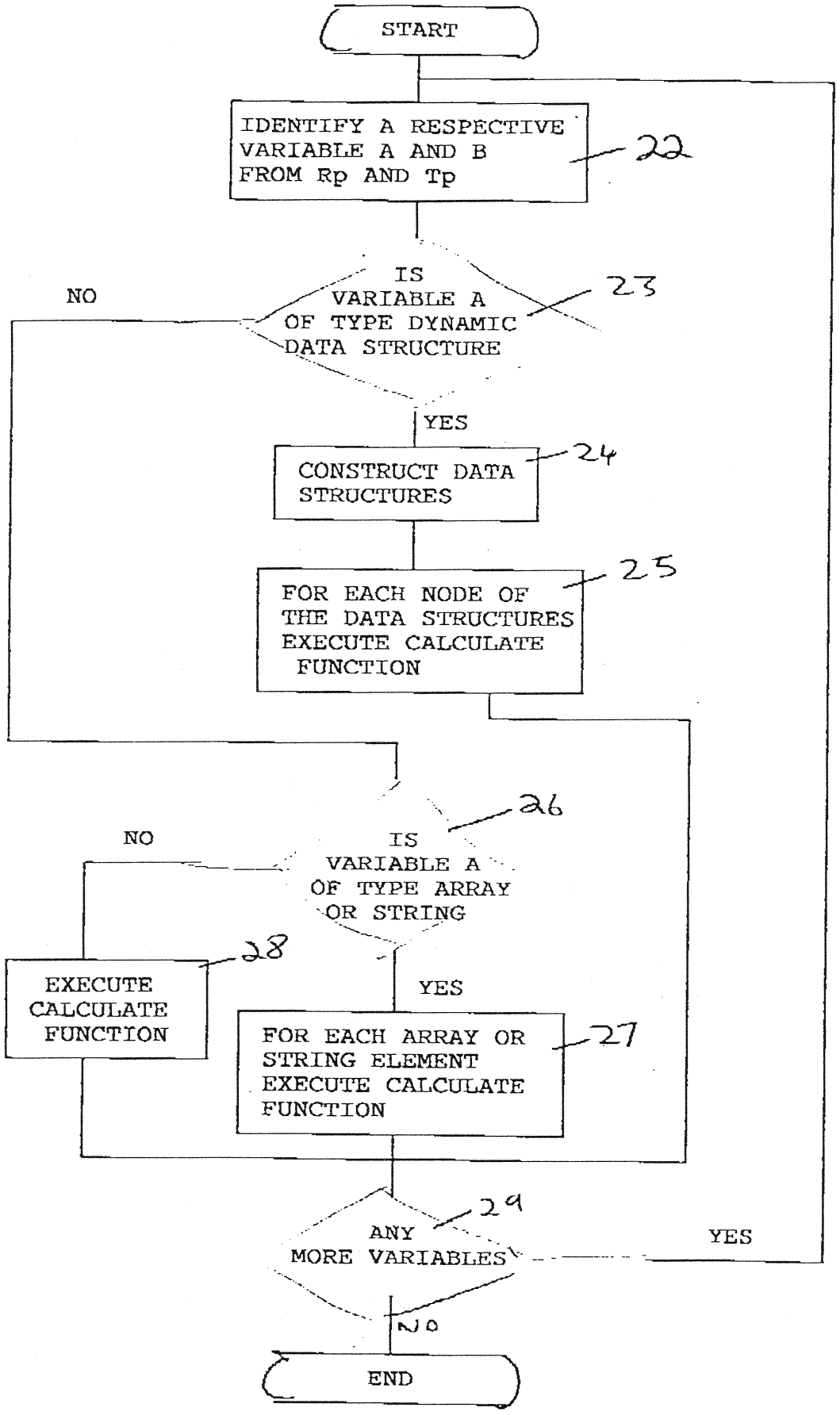

START
IDENTIFY A RESPECTIVE VARIABLE A AND B FROM Rp AND Tp
22
IS VARIABLE A OF TYPE DYNAMIC DATA STRUCTURE
23
NO
YES
CONSTRUCT DATA STRUCTURES
24
FOR EACH NODE OF THE DATA STRUCTURES EXECUTE CALCULATE FUNCTION
25
IS VARIABLE A OF TYPE ARRAY OR STRING
26
NO
YES
EXECUTE CALCULATE FUNCTION
28
FOR EACH ARRAY OR STRING ELEMENT EXECUTE CALCULATE FUNCTION
27
ANY MORE VARIABLES
29
YES
NO
END

FIG 3

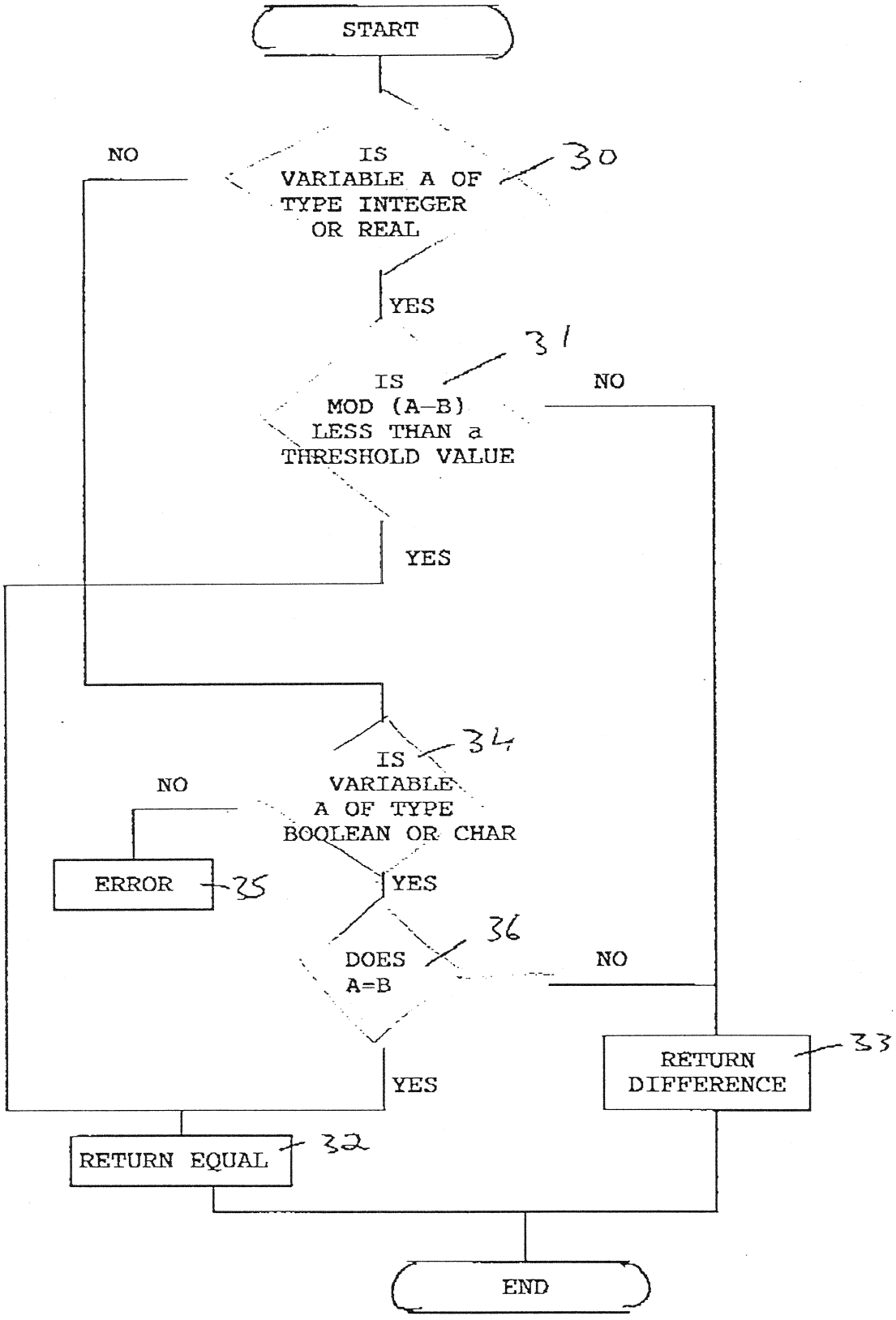

START
IS VARIABLE A OF TYPE INTEGER OR REAL
30
NO
YES
IS MOD (A-B) LESS THAN a THRESHOLD VALUE
31
NO
YES
IS VARIABLE A OF TYPE BOOLEAN OR CHAR
34
NO
ERROR
35
YES
DOES A=B
36
NO
YES
RETURN DIFFERENCE
33
RETURN EQUAL
32
END

FIG 4

```
test:= equal,difference;

function simple_types_equal(a,b,threshold_value) : test

begin

        if mod(a-b) < threshold-value then simple_types_equal := equal else

                simple_types_equal := difference;

end;

function one_dim_arrays_equal(a,b,threshold_value): boolean;

begin

        same := true;

        for i :=        max(low_bound(a),low_bound(b)) to
                        min(high_bound(a),high_bound(b)) do begin

                same := same and simple_types_equal(a[i],b[i],threshold_value)

        end;

        one_dim_arrays_equal := same;

end;

function multi_dim_arrays_equal(a,b,threshold_value): boolean;

begin

        same := true;

        for i :=        max(low_bound(a,1),low_bound(b,1)) to
                        min(high_bound(a,1),high_bound(b,1)) do

                for j :=        max(low_bound(a,2),low_bound(b,2)) to
                                min(high_bound(a,2),high_bound(b,2)) do

                        for k :=        max(low_bound(a,3),low_bound(b,3)) to
                                        min(high_bound(a,3),high_bound(b,3)) do

                                        .    .    .    .

                                             .    .    .    .

                        same := same and simple_types_equal(a[i,j,k,...],

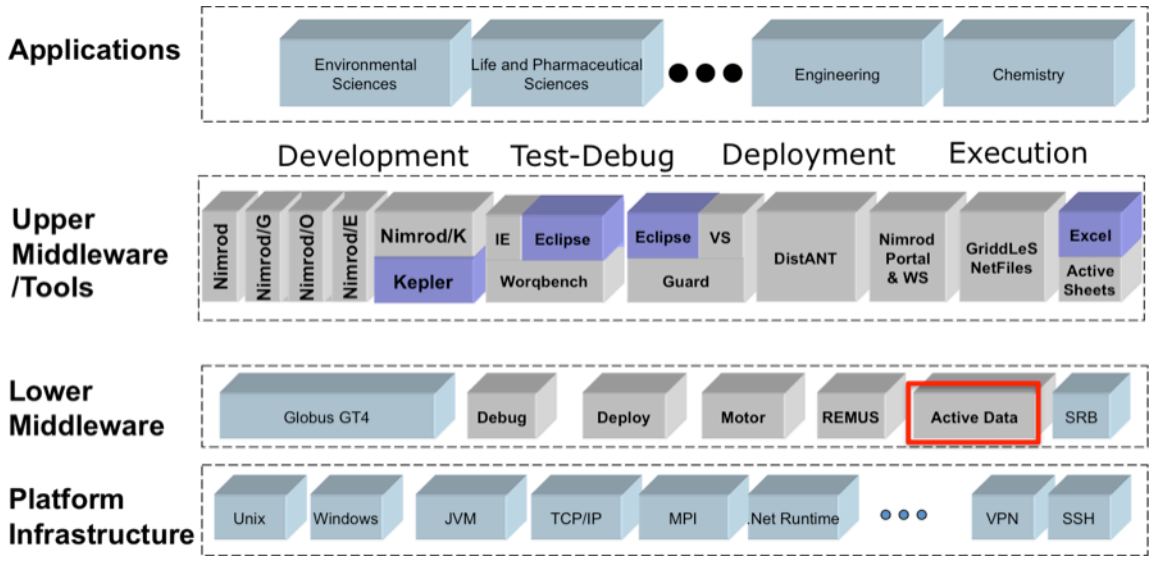
                        b[i, j,k,...],threshold_value)

        multi_dim_arrays_equal := same;

end;
```

FIG 5

Chapter 11: Active Data and Data Replication



This chapter discusses techniques for replicating data on the Grid, as explored in (50) and (51). We developed a framework, and a data life cycle called the Active Data model in (52).

(1) Ho, T. and Abramson, D. "The GriddLeS Data Replication Service", IEEE Conference on e-Science and Grid Computing, Melbourne, Dec 2005. 11-1 .. 11-8

(2) Ho, T. and Abramson, D. "A Unified Data Grid Replication Framework", 2nd IEEE International Conference on e-Science and Grid Computing. Dec. 4- 6, 2006, Amsterdam, Netherlands. 11-9 .. 11-16

(3) Ho, T. and Abramson, D., "Active Data: Supporting the Grid Data Life Cycle", ", 7th IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2007, pp 39 – 46, Brazil. 11-17 .. 11-24

Contributions

Ho was a PhD student under my supervision.

The GriddLeS Data Replication Service

Tim Ho and David Abramson
{tim.ho, davida}@csse.monash.edu.au

School of Computer Science and Software Engineering
Monash University,
900 Dandenong Rd,
Caulfield East, 3145, Australia

Abstract

The Grid provides infrastructure that allows an arbitrary application to be executed on a range of different computational resources. When input files are very large, or when fault tolerance is important, the data may be replicated. Existing Grid data replication middleware suffers from two shortcomings. First, it typically requires modification to existing applications. Second, there is no automatic resource selection and a user must choose the replica manually to optimize the performance of the system. In this paper we discuss a middleware layer called the GriddLeS Replication Service (GRS) that sits above existing replication services, solving both of these shortcomings. Two case studies are presented that illustrate the effectiveness of the approach.

1 Introduction

The Grid provides infrastructure that allows an arbitrary application to be executed on a range of different computational resources, providing significant flexibility for users. Scheduling decisions are often made at run time, and are based on metrics such as the availability of the computational resources, network bandwidth and any necessary data files as well as trying to match application characteristics to the most suitable platforms.

When the input files are very large, or when fault tolerance is important, users may distribute complete replicas of the data to multiple Grid nodes. Such data replication is often facilitated by special middleware such as the Storage Resource Broker from SDSC and the Replication Services from Globus. Whilst powerful, existing middleware suffers from two shortcomings. First, it typically requires modification to existing applications, which is problematic when large and complex legacy codes are executed. Many applications are too fragile to be readily adapted. Second, if the user wishes to optimize application performance, they must choose the replica manually as there is no automatic selection. This is complex and error prone, and also does not allow for performance changes that might occur during program execution.

"Grid enabling" legacy applications is not a simple task. Grid standards such as the Web Services Resource Framework (WSRF), as exemplified by the Globus Toolkit (GT4), offer APIs that provide a fairly low level of functionality. They also require fairly substantial modifications to existing codes. Accordingly, a number of systems, such as NinfG [24], P-Grade [18], Condor-G [22], GrADS [9], APST [10], Nimrod/G [6] and GriddLeS [5], attempt to bridge the gap between high level applications and the available Grid middleware.

In GriddLeS we provide a range of mechanisms that allow a legacy application to access local and remote files, and also other applications using pipes [4][5]. These mechanisms are built on conventional file access primitives, so whilst a code might "think" it is reading a local file, it could be accessing a remote file or even data that is being written by another application on another server. The system makes it possible to build flexible Grid pipelines involving more than one computational process.

We have developed an extension to GriddLeS to allow it to access replicated data, thus addressing the two shortcomings sited above. First, GriddLeS sits above existing data replication services and provides transparent access to files using conventional file access primitives. Thus, legacy codes do not need to be modified. Second, GriddLeS monitors parameters such as the instantaneous network bandwidth and resource availability, and dynamically chooses the most appropriate replica. In this way it can choose the most appropriate data replica at run time. Moreover, GriddLeS continuously monitors these parameters and, when replica data is read only, it switches sources, allowing it to adapt to dynamic changes in bandwidth. The GriddLeS Replication Service (GRS) uses the Network Weather Service to monitor the state of the network [20].

In this paper we demonstrate the effectiveness of the GriddLeS Replication Service using two case studies. One is a simple IO bound file copy, whilst the other is

computationally bound application (a regional weather model) running on an enterprise Grid. In both cases, the input files are available on subset of the grid nodes, and GriddLeS performs the mapping between the application and the data files transparently. We demonstrate that the system is able to adapt to changes in network bandwidth during the execution and that it is able to optimize the overall performance.

2 Data Replication Services

There are several Grid data replication systems already available. Grid Datafarm [17], initiated in Japan, is an architecture designed to handle hundreds of terabytes to petabytes of data using a global distributed file system. An implementation of this architecture, called Gfarm, has been developed. It manages the data distribution in the Gfarm file system by a metadata management system and the gfarm filesystem daemon. The metadata system contains the mappings from logical file names to the physical file names. A daemon provides access control to the remote files and monitors remote resources. A parallel I/O library is available for faster data analysis. Higher-level scientific applications can be constructed through the APIs.

The Globus Replica Location Service (RLS) is an implementation of the RLS framework [1] and is available as one of the components in the open source Globus Toolkit [12][31]. RLS is a distributed replica management system designed to replace the centralized Globus Replica Catalog in the previous version of Globus. In addition, the RLS is designed to tackle issues including scalability, reliability and security. The key feature of the RLS is to maintain and provide access to mapping information from unique logical file names to the physical copies. The locations of the data copies are recorded through the distributed registry service. RLS information is stored in an open source relational database such as MySQL [27] and PostgreSQL [28].

Besides the Globus RLS, there are other tools that are implemented based on the RLS framework, such as Reptor [15] and GDMP [2] from the European Union Data Grid Project [29] as well as the File Copy and Registration Service (CAR) [30].

The Storage Resource Broker (SRB) [3] is a client-server data management system. In addition to providing mechanisms to manage replicated datasets, SRB also allows programs to access data based on the attributes rather than filenames or physical locations. SRB employs a Metadata Catalog (MCAT) service which allows users to store metadata information for the datasets. The metadata is basically information that identifies and describes the associated data copies. Information contained in the metadata may include, but is not limited to, the physical location of the files, access control information and description of the data. SRB provides several different clients, including a set of command line tools called Scommands, a web portal called MySRB, and a Microsoft Windows client called inQ. There are also comprehensive APIs including C, C++, Java and Python.

SRB provides tools, such as srbIO and unixIO, to enable programs to access SRB data with small code modification. However, most of the replica systems require modification to an existing program that is originally not designed to support replicated data. Since code modification is error prone and is sometimes infeasible (e.g. no source code is available), there is a need for a tool that provides the existing software access to replicated data without code modification.

Further, certain levels of optimization should occur in order to minimize the program execution time. When a number of replicas are available, the application should decide which one is the best one to access and it should also switch to another one dynamically if a better one is found in run time.

In spite of these goals, very few of the replication systems provide optimization based on the resource status on the application level. Replica selection is done by the user and there is no support for dynamic replica selection during program execution. Some research aimed at providing mechanisms for replica selection [21][23][25] but currently there is little support for dynamic replica selection.

3 Introduction to GriddLeS

GriddLeS provides a general environment for the composition of Grid application from legacy code. The core concept in GriddLeS is GridFiles, a device which enables file based interprocess communication between software components. GridFile are supported by a component called the File Multiplexer (FM), as shown in Figure 1.

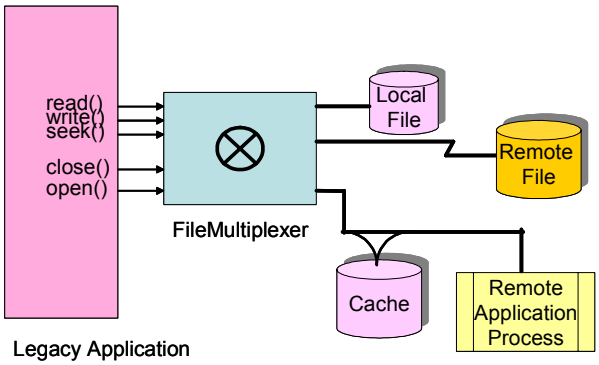


Figure 1 – The GriddLeS Architecture

The FM traps IO system calls from an application (e.g. open, close, read, write, etc.) and maps these operations to appropriate services dynamically. Originally the trapping mechanism is based on Bypass [7]. Since Bypass does not support more recent Linux distributions, Parrot is used as an alternative [8]. A Parrot GriddLeS IO driver redirects the IO calls to the corresponding GriddLeS IO routines.

When a program reads data, the FM can redirect the read operation to read from a local file, a remote file, or connect to a write operation on a local or remote machine using a shared buffer. These are processed by the Local File Client, the Remote File Client, and the Grid Buffer Client respectively, as shown in Figure 2. These three access modes make it possible to build flexible Grid workflows in which computations either read and write their data to and from files, or communicate directly via pipes. The latter allows writer and reader application to overlap IO and computation, and leads to efficient solutions when stream based computations are performed.

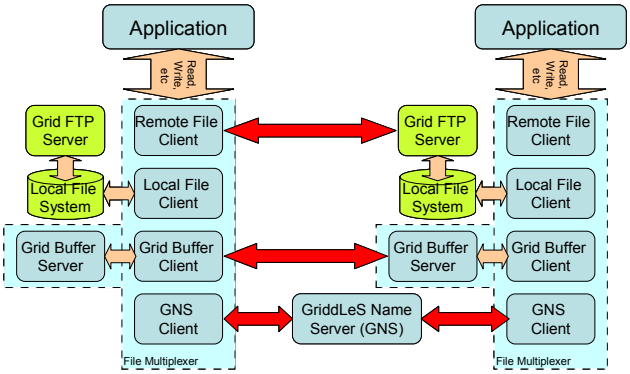


Figure 2 – Architecture of GriddLeS

The GriddLeS Name Service (GNS) is used for storing file mapping information that describes the configuration of a particular Grid application. The mapping is basically a key-value pair, containing the server name and the full path to the target file. This is a central place for the user to tell GriddLeS what to do when a program is trying to access a particular file (e.g. during the Open call). During runtime, GriddLeS reads the information and stores the values to a hash table. The mapping information can be changed at any time, and GriddLeS updates the information dynamically. Currently, the GNS is loaded based on the structure of a Grid workflow, and this can be specified using the Kepler tool [11].

The Local File Client is used to provide the running program access to local files. The Remote File Client provides access to remote files using GridFTP [26]. The Grid Buffer Client is required for inter-process pipelined communication. On the writer's side, it connects and sends written data to a remote Grid Buffer Server. On the reader's side it reads the data from the buffer server. Data is stored by the Grid Buffer Service in a hash table rather than a sequential file to support random access. If the data read by the reader application has not been written when a read is requested, the reader blocks until the needed data is available. When the data is read, it can be sent to a cache file, and the data in the hash table will be deleted. The cache file is important because it allows a reader to reread data from previous blocks (e.g. seek). The cache file also enables broadcasting, in which multiple readers are allowed to read data written by a single writer. In this case, the first reader retrieves the data from the hash table and subsequent readers get the data from the cache file. If no cache file is used, the Grid Buffer Client only allows sequential read and write operations. These additions make GridFiles significantly more powerful than conventional Unix pipes, that tyoically only operate on standard IO channels and also don't work across machine domains.

The potential effect of inter-process communication is significant because it allows overlapping between components that have file dependencies. This means that the execution can be greatly shortened as the programs can be run concurrently. Past case studies had proved the efficiency of GriddLeS with significant performance improvement [4][5].

4 The GriddLeS Replication Service

The existing GriddLeS environment described in section 3 provides significant benefits because it hides the complexity of data access on the Grid, regardless of whether it is stored in a local or remote file, or whether it is dynamically streamed between applications. However, the existing implementation does not support replicated data. To overcome this, a GriddLeS Replication Service (GRS) has been implemented.

The GRS provides mechanisms that allow an existing program to access replicated data without code modification. It is essentially a layer that sits between the legacy application and the storage resources, as shown in Figure 3.

The architecture of the GRS provides an abstraction of the underlying replica management systems. It provides a set of IO functions that hide all the replica system specific implementation from the application program. In fact, even the File Multiplexer does not require knowledge of the replicated data is accessed by the GRS.

Users can define mapping to a SRB replica in the GNS using this format:

```
<server>:<collection>:<object ID>:<resource>
```

The *server* is the SRB server and the *collection* is the SRB collection that contains the replica. *Object ID* is the identifier of a replica that is stored in the SRB server. The *resource* is the name of the SRB resource that contains the actual replica.

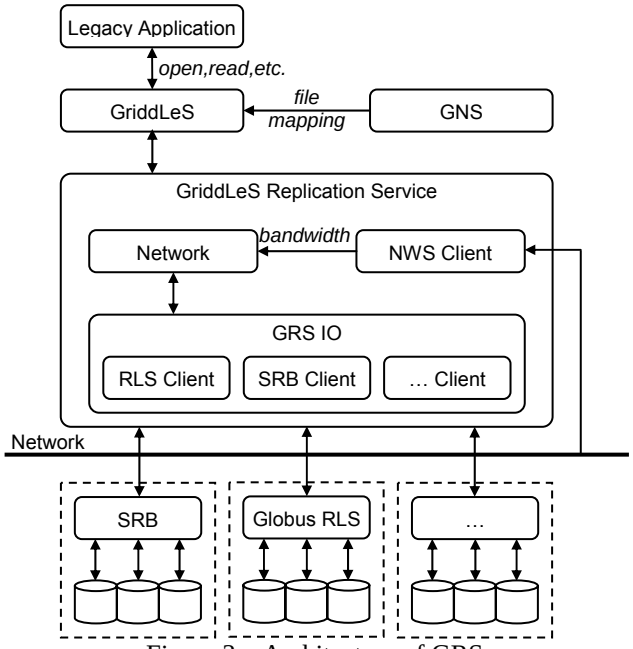


Figure 3 – Architecture of GRS

The GRS connects to a given SRB server using standard SRB login mechanisms, and thus the GNS does not require additional login information. This means that the .MdasEnv and .MdasAuth files must exist under the .srb directory. The former contains login information such as username, host address, host port and authorization scheme. The latter simply contains the password for login purpose.

Importantly, the GRS can choose which replica to address at run time, and this information can be hidden from the application. This provides a mechanism for optimizing the performance of the system by allowing GRS to not only choose an appropriate replica when the file is opened, but also to change that choice should network conditions change during the execution.

A NWS monitor is implemented as an important component of the GRS. This monitor is responsible for retrieving the network bandwidth information continuously when a SRB replica is opened. Bandwidth measurements between the SRB resources and the host running the program are retrieved periodically. Based on these measurements, the monitor will then forecast the bandwidth of each network and, finally, select and notify GRS the best resource for the next read call.

When an open call is intercepted, GriddLeS decides which file is to be opened based on the mappings in the GNS. If a SRB replica mapping is defined, GriddLeS passes the open call to the GRS. The GRS currently connects to the given SRB server using standard SRB password authentication. After the connection is made, instead of immediately opens the file, GRS communicates to the SRB server and tries to find out if there are other replicas exist in different servers. As a result, a list of all the replicas is generated. GRS then opens the replicas and maintains a list of opened replicas. This is important because the GRS must ensure at least one replica in the list is accessible. After the replicas are opened successfully, the network monitor executes in a separate thread and monitors the network status to each replica. Note that when replica exists in the localhost, no network monitor will be run and the local replica is used. The GRS normally uses the server with the best instantaneous and predicted bandwidth, thus server changes occur depending on the current network conditions. Importantly, this change is transparent to the running application.

Since each replica is located on a different server, and a server change may occur dynamically at anytime during execution, proper synchronization is required to avoid reading incorrect blocks of data when a server change occurs. Accordingly, each time a read is performed, the new offset of the replica is calculated and stored. Upon next read, if there is no change of the resource, then GRS simply calls the usual read routines. However, when there is a resource change, GRS seeks to the correct offset before reading from the replica.

When closing, the GRS closes all the opened replicas as well as the connections to them. Also, the network monitor is killed.

Currently, the GRS only supports access to SRB datasets but we plan to support other replica systems in the near future, including the Globus RLS. The following IO functions are provided by GRS:

*int grs_open(const char *path, int flags, mode_t mode);*
*ssize_t grs_read(int fd, void *buf, size_t count);*
*ssize_t grs_write(int fd, const void *buf, size_t count);*
*int grs_stat(const char *path, struct stat *buf);*
*int grs_fstat(int fd, struct stat *buf);*
off_t grs_lseek(int fd, off_t offset, int whence);
int grs_close(int fd);

These functions have the same arguments and return values to the corresponding system calls. When replica functions are required, GriddLeS simply forwards the arguments from the system calls to the GRS IO routines. The *grs_open* function returns a GRS-specific unique replica identifier instead of a normal system file descriptor. This identifier allows the GRS to identify the associated replicas in the subsequent IO calls.

5 Case Studies

In this section we present two case studies, one IO bound and the other compute bound, to illustrate the functionality and performance of the GRS. The first of these involves a simple program that takes a copy of the replicated data, whilst the second is a regional weather model in which the input files are replicated across a number of servers. *Importantly, neither of these applications required source changes because GriddLeS hides the details of the file replication system.*

5.1 The testbed

Table 1 shows the configuration of our testbed, consisting of 10 grid resources. All machines are running SRB 3.2.1 and NWS 2.10.1, with one SRB MCAT server running on *argos*, which stores the mappings and replica details, and non-MCAT servers on the others. The testbed spanned a number of machines within our department, but also machines in the UK, Malaysia, US and Hong Kong.

Machine	**Details**
argos.csse.monash.edu.au	Pentium IV 2.4GHz 480MB RAM Linux
globus.csse.monash.edu.au	Pentium II 333MHz 128MB RAM Linux
mosca.csse.monash.edu.au	SPARC SUNW, Ultra-5_10 Solaris
ocean.csse.monash.edu.au	Pentium III 500MHz 384MB RAM Linux
remus.dstc.monash.edu.au	Pentium IV 2.4GHz 512MB RAM Linux
romulus.dstc.monash.edu.au	Pentium IV 2.4GHz 512MB RAM Linux
aurora.cs.usm.my	Pentium III 1.4GHz 2048MB RAM Linux
bouscat.cs.cf.ac.uk	Pentium III 1Ghz 1536MB RAM Linux
freak.ucsd.edu	Athlon XP 1800+ 768MB RAM Linux
gideon.cs.hku.hk	Pentium IV 2.26GHz 1024MB RAM Linux

Table 1 – Testbed configuration

5.2 Methodology

For each experiment, we replicated the input files to the 9 non-MCAT servers. Each experiment was run five times on *argos* and an average time was then calculated and compared. We compare the execution time on reading from a local file, reading from a remote file at two different levels of bandwidth, as well as the improvement from the dynamic replica selection.

To evaluate the improvement from the optimizations, the bandwidth between the program running host and replica servers was lowered by transmitting large amount of data using `scp` to the servers from a separate machine outside the testbed. Since *globus* and *romulus* have the highest bandwidth, we lowered their bandwidth randomly in order to see determine the effect of the network disruption.

5.3 Replicated file copy application

This case study considers a simple file copy program that reads data from one server and copies it to another machine. Multiple copies of the input file are available and the system chooses the most appropriate one at any time based on NWS statistics and predictions for the bandwidth.

Since the data is read continuously, the network conditions will greatly affect the performance if the input data is located remotely. We used a 140MB file as the input to the file copy program and the results are shown in Table 2.

Bandwidth	**Data read from**	**Average time mm:ss**	**Performance Change**
~8 Mbps	romulus	03:02	100%
~0.2 Mpbs	aurora	29:28	971%
Variable	Variable	03:32	116%

Table 2 – details of the file copy experiments performed

When the file copy program reads data, GriddLeS intercepted the call and passes the call to the GRS, which reads the data from the best SRB resources selected by the network monitor. Since the data is read across the network, the execution time was increased from the original 9 seconds to 3 minutes. When the bandwidth between the host and the SRB resource was reduced significantly, the execution time was further increased to more than 29 minutes.

The last experiment evaluates the improvement gained by using dynamic resource selection in the GRS. The bandwidth between the host and the SRB resources were lowered randomly. The network monitor informs the GRS

when it finds a SRB resource with better network bandwidth. During run time, the GRS switches to a better server based on the decision made by the network monitor. The result shows that the performance is nearly as good as when there is no network contention.

5.4 A regional weather model

This application concerns a computationally intensive regional weather model called DARLAM [13][14]. DARLAM. Written in Fortran, the program computes temperature and wind vectors by solving a numeric model of the atmosphere. It reads data from a 190MB input file (darlam_input.nc). This input file is replicated and stored in 9 SRB resources. A SRB MCAT server running on *argos* stores the locations of the replicas. A SRB file path is defined in the GNS to redirect the system call to read from a SRB replica instead of a local file. The results are shown in Table 3.

Exp	Model runs in	Data read from	Ave time mm:ss	Perf Change
1	argos	argos	38:39	100%
2	argos	romulus	41:47	108%
3	argos	aurora	77:32	200%
4	argos	Variable	45:02	116%

Table 3 – details of the DARLAM experiments performed

In experiment 1, DARLAM was run without GriddLeS, and the input data was stored locally. The other three experiments read replicated data from SRB resources using the GRS. In all experiments, DARLAM was run on *argos*. The IO calls made by DARLAM were intercepted and replaced by the corresponding GriddLeS IO routines.

In experiment 2, DARLAM was run on *argos* and the input data was read across the network. In this case, when DARLAM opens "darlam_input.nc", GriddLeS passes the open call to the GRS. The GRS inspects the MCAT to obtain a list of available replicas. As expected, the execution time was slightly longer than in experiment 1 because the data was sourced across the network. However, unlike the first case study, because the application was predominantly compute bound, the time only increased by about 8%.

In experiment 3, we redirected all the IO calls to *aurora*, which has a very low bandwidth (~0.2Mpbs). This time, because of the dramatic loss of network bandwidth, DARLAM was over 100% slower than in experiment 1.

In experiment 4, we replicated the input file across the 9 SRB resources and the bandwidths between *argos* and the resources were randomly lowered. As observed the execution time in this experiment was only slightly longer than experiment 2 in which no bandwidth perturbation was performed. This is because the GRS was dynamically switching to the most effective server, as is illustrated in Figure 4. The upper part of Figure 4 shows which server was used over time and the lower part of the figure shows the bandwidth data retrieved from the NWS between *argos* and the other machines. The graph only shows 5 local machines because the other 4 external machines have very low bandwidth (<1Mpbs) and thus during runtime no data was read from them. The results clearly shows that the GRS is able to change the data source dynamically.

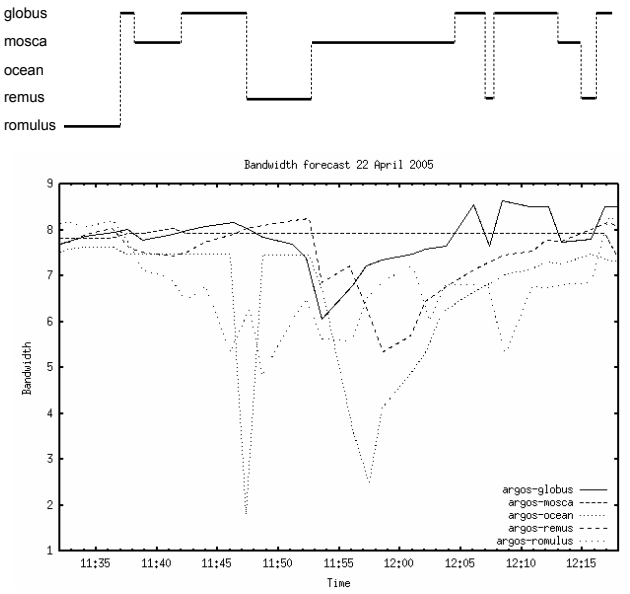


Figure 4 – Server selection against time

6 Conclusion

This paper has discussed a GriddLeS Replication Service (GRS) which allows existing applications to access replicated data. Importantly, this is performed without any modification to the application. We have described the architecture of the GRS, which provides a set of IO routines for access to different replica management systems, and have illustrated our implementation which only currently supports the SRB from SDSC. This abstraction avoids code modification because the application does not need to have specific code for any particular replica systems. In addition, the GRS provides certain level of optimization by dynamically switching to a better server. This is done by monitoring the network condition using the Network Weather Service at runtime and forecasting which connection has better bandwidth. The server selection can occur at any time during program execution to tackle unreliable network connection. This

improves fault tolerance and provides a reliable data input because the same piece of data is available in several servers. We demonstrated that by changing the data server based on the network condition can increase the overall performance.

The network monitor in the GRS currently only takes into consideration the connection bandwidth. We plan to extend it to take incorporate other resource information such as network latency. This is important, because a high latency network will lower the server response time even the bandwidth of the server is high. Load on the resource should also be considered because a high workload host may also have lower response time. Further, instantaneous network status may need to be considered because when a server suddenly goes offline, the GRS may not be able to react quickly due to the lag in the NWS client.

Moreover, we plan to support further performance optimizations such as data striping [19] and data prefetching [16]. Data stripping allows parts of the dataset to be stored across the Grid for better performance. In this case, GriddLeS could provide facilities for the user to define which portions of the data are required. To further reduce the processing time, the GRS could employ data prefetching, in which the data is always read ahead and stored locally (e.g. in the memory). This can greatly eliminate the effect of high network latency. Finally, we plan to support other data replication systems such as the Globus RLS.

Acknowledgments

This project is supported by the Australian Research Council and Hewlett Packard as part of an ARC Linkage project. We wish to acknowledge our colleagues at the CSIRO, in particular John McGregor for providing access to the DARLAM model, and Jagan Kommineni who built much of the GriddLeS library. Thanks also go to the SRB team at SDSC for their assistance. Tim Ho is supported by an Australian Postgraduate Research Award (APRA-I).

References

[1] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunst, M. Ripeanu, B, Schwartzkopf, H, Stockinger, K. Stockinger, B. Tierney, “Giggle: A Framework for Constructing Scalable Replica Location Services”, Proceedings of Supercomputing 2002 (SC2002), November 2002.

[2] A. Samar and H. Stockinger, “Grid Data Management Pilot (GDMP): A Tool for Wide Area Replication”, IASTED International Conference on Applied Informatics (AI2001), Innsbruck, Austria, February 2001.

[3] C. Baru, R. Moore, A. Rajasekar and M. Wan, “The SDSC Storage Resource Broker”, Proceeding of CASCON'98 Conference, Toronto, Canada, 1998.

[4] D. Abramson and J. Komineni, “A Flexible IO Scheme for Grid Workflows”, IPDPS-04, Santa Fe, New Mexico, April 2004.

[5] Abramson, D., Kommineni, J., McGregor, J. and Katzfey, J. “An Atmospheric Sciences Workflow and its Implementation with Web Services”, Future Generation Computer Systems, 21 (2005), pp 69 – 78.

[6] D. Abramson, J. Giddy and L. Kotler, “High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?”, International Parallel and Distributed Processing Symposium (IPDPS), pp 520- 528, Cancun, Mexico, May 2000.

[7] D. Thain and M. Livny, "Bypass: A tool for building split execution systems", Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing, Pittsburg, Pennsylvania, pp.79-85, August 1-4, 2000.

[8] D. Thain and M. Livny, "Parrot: Transparent User-Level Middleware for Data-Intensive Computing", Workshop on Adaptive Grid Middleware, New Orleans, Louisiana, September 2003.

[9] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, “The GrADS Project: Software Support for High-Level Grid Application Development”, International Journal of High Performance Computing Applications, vol. 15, no. 4, pp. 327-344, Winter 2001.

[10] H. Casanova, G. Obertelli, F. Berman and R. Wolski, “The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid”, Proceedings of Super Computing 00, Dallas, Texas, November 2000.

[11] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher and S. Mock, "Kepler: An Extensible System for Design and Execution of Scientific Workflows", 16th International Conference on Scientific and Statistical Database Management (SSDBM'04), Santorini Island, Greece, 21-23 June 2004.

[12] I. Foster and C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit”, International Journal of Supercomputer applications, vol. 11, issue 2, pp. 115-128, 1997.

[13] J. L. McGregor, K. C. Nguyen and J. J. Katzfey, “Regional Climate Simulations using a Stretched-grid Global Model”, Research Activities in Atmospheric

and Oceanic Modeling, H. Ritchie (ed.), pp. 3.15-3.16, 2002.

[14] J. K. McGregor, K. J. Walsh and J. J. Katzfey, “Nested Modelling for Regional Climate Studies”, A. J. Jakeman, M. B. Beck and M. J. McAleer (eds.), Modelling Change in Environment Systems, J. Wiley and Sons, pp. 367-386, 1993.

[15] L. Guy, P. Kunszt, E. Laure, H. Stockinger, K. Stockinger, “Replica Management in Data Grids”, Technical Report, Global Grid Forum - GGF5, Edinburgh, Scotland, July 2002.

[16] M. S. Pérez , J. Carretero, F. García, J.M. Peña and V. Robles. "MAPFS: A Flexible Infrastructure for Data-Intensive Grid Applications", Proceedings of the first European Across Grids Conference, Febrero 2003, Spain.

[17] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, H. Sato, Y. Tanaka, S. Sekiguchi, Y. Watase, M. Imori, T. Kobayashi, “Worldwide Grid Data Farm for Petascale Data Intensive Computing”, Technical Report, Electrotechnical Laboratory, ETL-TR2001-4, 2001.

[18] P. Kacsuk, J.C. Cunha, G. Dozsa, J. Lourenco, T. Antao and T. Fadgyas, “GRADE: A Graphical Development and Debugging Environment for Parallel Programs”, Parallel Computing Journal, Elsevier, vol. 22, no. 13, pp. 1747-1770, February 1997.

[19] R. Marco, J. Marco, D. Rodriguez, D. Cano, and I. Cabrillo. "RAID-1 and data stripping across the GRID", Lecture Notes in Computer Science, 2970:119-123, March 2004.

[20] R. Wolski, N. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing", Journal of Future Generation Computing Systems, Vol.15, No.5-6, pp.757-768, October, 1999.

[21] S. Vazhkudai, S. Tuecke, I. Foster. “Replica Selection in the Globus Data Grid”, Proceedings of the first IEEE/ACM International Conference on Cluster Computing and the Grid (CCGRID 2001), IEEE Computer Society Press, May 2001.

[22] T. Tannenbaum, D. Wright, K. Miller and M. Livny, “Condor – A Distributed Job Scheduler”, Thomas Sterling, editor, Beowulf Cluster Computing with Linux, The MIT Press, 2002.

[23] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini. “Design of a Replica Optimisation Framework”, Technical Report.,DataGrid-02-TED-021215, CERN, Geneva, Switzerland, 2002.

[24] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura and S. Matsuoka. "Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing", Journal of Grid Computing, Vol.1 No.1, pp.41-51, 2003.

[25] Y. Zhao, and Y. Hu."GRESS - a Grid Replica Selection Service", ISCA 16th International Conference on Parallel and Distributed Computing Systems, PDCS-2003.

[26] Globus: GridFTP Protocol and Software, http://www.globus.org/datagrid/gridftp.html.

[27] MySQL, http://www.mysql.com.

[28] PostgreSQL, http://www.postgresql.org.

[29] The DataGrid Project, http://eu-datagrid.web.cern.ch/eu-datagrid/.

[30] The File Copy and Registration Service, http://www.isi.edu/~shishir/car/.

[31] The Globus Toolkit, http://www-unix.globus.org/toolkit/.

A Unified Data Grid Replication Framework

Tim Ho and David Abramson
{tim.ho, davida}@csse.monash.edu.au

Monash e-Science and Grid Engineering Lab
Faculty of Information Technology
Monash University,
900 Dandenong Rd, Caulfield East, 3145, Australia

Abstract

Modern scientific experiments can generate large amounts of data, which may be replicated and distributed across multiple resources to improve application performance and fault tolerance. Whilst a number of different replica management systems exist, particular communities usually adopt a single system. This creates problems when an application program spans more than one community, because it may need to target more than one middleware layer. One solution to this problem is to build a more flexible data access layer above the specific replica middleware. In this paper, we discuss such an architecture, the Grid Replication Framework, which provides applications with an abstract interface to existing replica systems. Further, the framework's flexible plug-in architecture makes it easy to support new middleware as it becomes available.

1. Introduction

Grid middleware enables the sharing of geographically distributed resources, such as high performance computers, data archives and scientific instruments, and provides mechanisms that unify access by hiding much of the inherent heterogeneity [7]. Such middleware not only provides access to high-end computational data and instrumentation services, but also allows users to perform their tasks with little regard to geographic location.

Scientific applications, such as high energy physics [23][25], geophysics [21][26] and astronomy [24][27], often generate large amount of data, and users who require access to the data are sometimes geographically dispersed. Such data can be replicated and stored in different locations in order to optimize data access performance and provide better fault tolerance. Data replicas can significantly reduce the data access latency, and hence improve the program performance.

Data replication in the Grid has attracted a great deal of interest [1][2][3][10][12]. In general, a replica management system provides mechanisms to search, locate and access existing data, as well as to create and register new data. It also defines how multiple data replicas can be identified, and how the user may access such data. The latter may consist of different access level and/or security features (e.g. user certificates) in order to control who is allowed to access the data. There are a few research projects [15][18][19] that explore optimal methods for choosing which replica to access. These systems rely on various metrics, such as network conditions, distance between the application and the data, dataset size, average file transfer time, and overall performance of the remote servers.

Several Grid-based data replication systems exist, including Grid Datafarm [12], Globus RLS [11] and SRB [3], to name a few. These systems each have their advantages and disadvantages, but have been widely adopted by specific communities. One common shortcoming is that they use different protocols for accessing data, and thus, clients for one replica system usually cannot easily access data stored by another system. In this case, third party tools, such as GridFTP [17] distributed in the Globus Toolkit [28], are often required to move the data across the network.

This drawback can be significant. Imagine a scientific project that has distributed terabytes of experimental results using, say, the SRB. Later, the project is involved in a collaboration based on Gfarm. In order to share and analyse the data, the users of both projects have several choices. They may agree on using one replica system and move all data to it, or they can employ third party file copy tools, and/or modify the scientific applications to read data from both systems. Moving data may be expensive and can lead to incoherence problems when the data is updated. Moreover, a program may only require access to small portion of a large file, in which case copying the entire file can be inefficient. On the other hand, code modification tends to be error prone. This also means that users of one system must also have knowledge of the other. Clearly, a higher level middleware that provides an abstract interface to the different replica systems would offer significant flexibility for application developers.

We have developed a Grid Replication Framework (GRF) that unifies and provides a common API to a

number of current replica management systems. The GRF has a flexible architecture so that users can add new clients for other replica systems without modifying the framework itself. The API builds on existing IO primitives and provides applications transparent access to the replicated data stored in supported replica servers. When an application is developed on top of GRF, users specify which replicated data should be accessed by placing some meta data in a mapping file. Thus, without modifying the application code, the user can alter the path to the dataset and instruct GRF to read data from any supported replica servers. We have demonstrated the effectiveness of the GRF with our own GriddLeS Replication Service [5][16]. By leveraging GriddLeS over the GRF, applications can access a range of different replica management systems transparently without source code modification.

In addition to providing a single uniform API, the GRF also provides optimization mechanisms for improving the data access performance. It does this by monitoring the network bandwidth and latency to the various replica servers, using a tool like the Network Weather Service (NWS) [14], and it then dynamically chooses the most appropriate replica during execution.

In this paper, we discuss the components of the GRF, and its plug-in architecture. We also demonstrate that the framework has a flexible design to support new replica systems.

2. Access to replicated data

As mentioned, scientific data is often replicated and distributed. Managing replicated data is a non-trivial task Thus, several replica management systems have been developed to tackle issues such as user access control, scalability, fault tolerance and performance.

The Storage Resource Broker (SRB) [3] is a data management system that employs a client-server architecture. It provides mechanisms to access data based on some attributes other than filenames. SRB uses a MCAT (Meta data Catalog) service to store meta data information for the stored datasets. Such information is used to identify and describe the associated data. Users of SRB can access the data using the command line clients called Scommands. Other means including a web portal called MySRB, and a Windows-based client called inQ. There are also APIs in C, Java and Python. SRB also provides two libraries, SrbIO and UnixIO, to facilitate code modification for legacy programs.

Initiated in Japan, Gfarm is an implementation of the Grid Datafarm architecture [12]. This architecture is designed to handle hundreds of terabytes to petabytes of data using a global distributed file system. Different from SRB, Gfarm focuses on a Grid file system that provides scalable IO bandwidth and scalable parallel processing by integrating many local file systems and clusters. It uses a meta data management system to manage the file distribution, file system meta data and parallel process information. A file system daemon runs in each node to support remote file operations. The daemon also handles user authentication, file replication, node resource control and status monitoring. Files stored in the Gfarm file system can be accessed using a C library, or the Gfarm command line clients. Also, a syscall-hook library is provided for legacy programs to access Gfarm files.

The Globus Replica Location Service (RLS) [11] is an implementation of the RLS framework [1]. RLS is designed to replace the centralized Globus Replica Catalog in previous version of the Globus Toolkit. The RLS maintains and provides the mapping information from unique logical file names to physical file names; each of the physical file names usually contains the actual location of a file. It provides API and client commands to register, query and remove the file mappings but does not provide any data access mechanism. Other tools are needed in order to access the data. Files registered in the RLS can be copied using other tools such as GridFTP [17] and, as a technical preview, the Data Replication Service (DRS) [20]. Despite this, there is no support provided by RLS to perform simple file IO.

For most applications data access usually involves conventional file operations, such as open, read, write, seek, stat and close. Each of the above three systems are built using different standards and protocols. Therefore, data access to one system is different from the others.

SRB and Gfarm both provide APIs for accessing replicated data and the APIs are in Unix IO style. Thus, performing file IO on an SRB object or a Gfarm file is similar to performing local file IO. However, this can create problems in research collaboration when shared data are stored in different systems, because an application developed to read SRB data cannot read files stored in Gfarm. Clearly, a more flexible data access middleware on top of the systems is needed.

Such middleware should provide transparent data access to the application. This means that the application is not required to know the location of the file or how to access it. An example of such middleware that has similar goals is GFAL [22], which is a library that presents Unix IO style interface for IO operations to provide file access using logical file names rather than fixed file names. However, one significant disadvantage in GFAL is that it does not support multiple replica systems, including SRB and

Gfarm, which means that the application cannot access data stored in different replica systems.

GFAL has another limitation, namely, no optimisation is performed during file access. Optimised data access can significantly increase the execution performance, as presented in [16]. Optimisation methods varies, but the goal is to locate a better data source using information such as network condition, CPU load, distance to the data, etc.

3. The Grid Replication Framework

GRF unifies the existing replica systems and hides the underlying complexity of the different data access mechanisms. Our current implementation supports , SRB, Globus RLS and Gfarm. Building on top of GRF, a Grid application is provided with a set of Unix style IO functions that support transparent data access. The file IO operations are separated from the application itself, so that there is no need to modify the application code when change of location occurs.

Unlike the GriddLeS Replication Service, also implemented by our research group, the GRF has a flexible modular design for supporting new replica clients. With this plug-in design, users can easily develop clients for GRF that add data access supports for other systems.

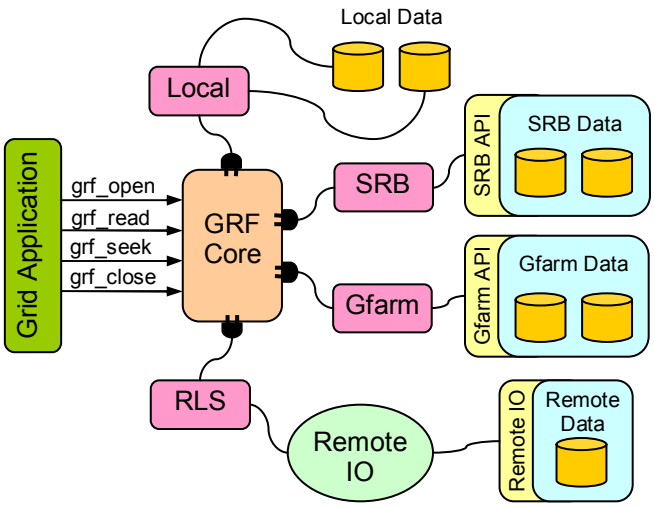


Figure 1 Architecture of GRF

3.1 Architecture of the framework

The GRF consists of a number of plug-ins that provide data access mechanisms for each replica system. As shown in Figure 1, a Grid application uses the GRF interface to perform desired IO operations. This interface abstracts the interfaces to the replica systems so that the application is not required to know about where the data is located, and how the data can be accessed.

In fact, the core of GRF does not implement any file access mechanism. Instead, the actual file operations are provided by replica client plug-ins. Currently, there are four plug-ins developed: local file access, SRB replica access, Globus RLS replica access, and Gfarm replica access. These plug-ins are used to provide system-dependent data access to the various replica systems. Without proper data access plug-ins, GRF will not perform any IO operations.

The framework consists of several components, including an IO interface, two configuration files, a Data Selection Service (DSS) and plug-ins that provide system-specific file access. Figure 2 shows the major

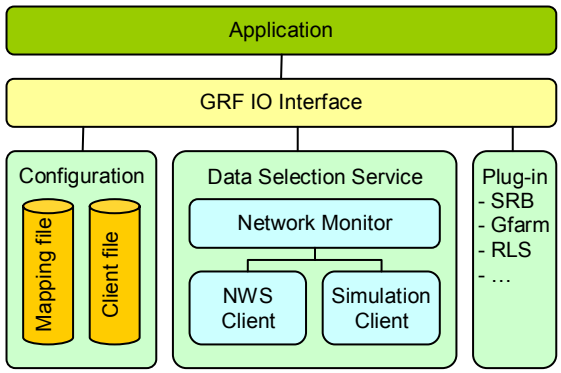


Figure 2 Components of GRF

components in the framework.

The GRF IO interface consists of a number of routines for performing IO on supported systems. The current implementation supports *grf_open*, *grf_read*, *grf_write*, *grf_stat*, *grf_seek* and *grf_close* operations.

The mapping file contains the mappings from keys to the physical paths of the files. A key is a unique identifier that serves as a logical file name used by the Grid application. During the *gfs_open* call, the GRF uses the key given by the application to find the data location from the mapping file. Below is an example (on one line) that instructs GRF to access an SRB object, where *input* is the key.

```
myhost.monash.edu.au:input
srb:srbhost.monash.edu.au:5544:/mona
sh-srb/home/tim.srb:input.dat
```

Each entry in the mapping file consists of two main parts separated by an empty space. The first part is an identifier that matches the key used in the *grf_open* call. The second part consists of a client key used by GRF to identify which plug-in is used for the IO;

information after the first colon is system-specific and is parsed and used by the plug-in. The client key refers to the library path information stored in the client file. The GRF examines the client file and finds out the full path to the plug-in shared library. The library is loaded during runtime for any supported IO operations and the system-specific file details are passed to the corresponding routines. An example of an entry in the client file is as follows:

```
srb=/home/tim/grf/io/libsrb.so
```

The mapping file and the client file are two key components because they contain the mapping to the actual data location as well as the client library location. When change of file location occurs, the user only modifies the mapping file, rather than changing the application code. When a new plug-in is added, only the client file requires changes. Avoiding code modification is a major advance to the conventional IO routines and provides data access flexibility to the application running on the Grid.

The GRF employs a Data Selection Service (DSS) for the read operations. This service uses a network monitor to continuously monitor the network condition to the data servers and dynamically switch to a better one based on network statistics. This network monitor includes a NWS client that retrieves the NWS measurements and generates network condition forecasts. When a better server is found based on the NWS predictions, GRF will dynamically switch to, and source the data from, the better one. Because this selection is made dynamically during runtime, the program execution time can be significantly shortened. DSS is an optional feature and can be disabled when no optimisation is required.

Since the file offset information for replicas are stored independently in each replica server, change of replica source may result in accessing incorrect blocks of data. To avoid this, the DSS employs proper synchronization for file access every time when a replica selection is done. The file offset is stored and updated in GRF at the end of each IO operation. If there is a change to the server, then GRF will seek to the correct offset before accessing the replica.

For evaluation purposes, simulated network statistics can also be used. Apart from real time NWS forecasts, the network monitor can also use a user-defined network statistics for simulation and evaluation purposes. Currently, network bandwidth is used for deciding which replica server is used. When needed, users can define a sequence of network bandwidth in a file and the DSS will select a server based on the information provided in that file.

3.2 Data access mechanisms

The current GRF implementation supports four different access mechanisms: local file access, SRB access, Gfarm access and RLS access. Each of them provides specific file IO for the corresponding system.

3.2.1 Local file access

There are two cases where the application using GRF may access a local file. The first is when the user wants to use conventional IO for local files. In this case, a full path beginning with a forward slash ("/") to a local file should be specified in *grf_open*.

The other case is when a key is specified but the mapping actually refers to the local file plug-in. The reason of doing so varies. One reason is that the user wants to take advantage of the flexibility of GRF because change of file location may occur. This can happen when the application requires access to a replica that is copied to the local machine for better performance. It also benefits if the application requires frequent change of file location.

In either case, conventional IO routines will be used. This is a straightforward task because the arguments and return values of the GRF IO are similar to the conventional one. Yet, there are two major differences.

The first difference is the file descriptor (FD) used in the two cases. For the first one, the FD is exactly what the operating system returns. Basically, when a file path beginning with a slash is specified, the GRF simply passes the arguments to the corresponding Unix IO routines and returns what the routine returns.

However, for the second situation, a GRF-specific FD will be used. The main reason is that in subsequent IO operations GRF requires this FD to find out whether the operations are GRF IO or Unix IO. In addition, this GRF-specific FD is a simple integer that emulates the FD normally returned by the *open* function.

Another difference is that when a GRF client library is used, a GRF-specific replica object is created. The object is used to store information including the client module, the key for the mapping, the attached network monitor, the file current offset, and most importantly a handle to a replica system-specific object that contains information such as the server location, port, path, etc. The replica object is identified later using the FD.

To use the local file access plug-in, a local file entry should be added to the mapping file. This looks like the following:

```
myhost:input local:/tmp/input.dat
```

In this example, the host address (where the program running in) and the key identifier are on the left hand

side, where *input* is the key that would be specified in the *grf_open* call. On the right hand side, *local* is the key to identify the client library. The other information on the entry is the full path to the local file. All file operations using *input* as the key will be performed on this file.

3.2.2 SRB replica access

A SRB plug-in, which has similar functions to the one used previously in GriddLeS [16], is also implemented. When an application requires access to data stored in the SRB, the current implementation of the SRB client plug-in uses the standard SRB user authentication (i.e. requires .MdasEnv and .MdasAuth) to connect to the specified SRB server. When the server is connected, the client will obtain a list of available replicas from the SRB server. The GRF then creates an object that represents the list of replicas and proceeds to open the SRB files. When required, a network monitor is started and attached to the GRF object. At the end of the open function, *grf_open* returns a unique GRF-specific file descriptor (FD) that emulates the FD normally returned by the conventional *open* function.

Below is an example of an entry in the mapping file that uses the SRB plug-in. The key identifier on the left hand side is omitted.

```
srb:mySrbhost:5544:/srb/home/tim.srb:
input
```

On this entry, the word *srb* refers to the SRB client library and other required information includes the SRB server address, port, path to object, and the object identifier. Since this information is specific to SRB, it is the job of the SRB plug-in to parse this information. GRF does not have knowledge about the details and it merely passes this to the SRB plug-in.

3.2.3 Gfarm replica access

A Gfarm plug-in provides a GRF interface for the Gfarm file system. Similar to SRB, Gfarm provides well defined C interfaces that have IO functions similar to Unix-style IOs for performing remote file operations. Since the Gfarm execution environment must be initialized before any file operation, the first task in the open call is to initialize the Gfarm environment. After that, the file path (i.e. Gfarm URL) is used with the Gfarm open routine, *gfs_pio_open*. The returned Gfarm file handle is stored in the GRF replica object and this replica object will be stored within GRF for subsequent IO operations. Usual Gfarm user authentication is used; the configuration file .gfarmrc is required.

The Gfarm plug-in does not use GRF's replica selection feature because when a Gfarm file is accessed, the Gfarm file system daemon automatically selects the best replica based on average CPU load and response time. Thus, optimization is done by Gfarm.

The following is an example of an entry in the mapping file for accessing a Gfarm file:

```
myhost:input gfarm:input.dat
```

In this case, when the application is opening a file using the key *input*, GRF passes the file information to the Gfarm plug-in and perform the IO operations on the Gfarm file.

3.2.4 Globus RLS replica access

The Globus RLS plug-in for GRF is considerably different from the other two discussed to date, because RLS lacks a data access API. Although files registered in RLS can be copied using other tools, a set of Unix-like IO routines would be preferred when an application requires access to a file (probably a small portion), and to perform IO operations, such as stat and seek. This is especially true for a workflow that does file-based (and potentially replica-based) communication on the Grid, when no file copy is desired.

As a result, a simple remote IO interface that provides file IO access to the remote data is implemented in the RLS plug-in. This interface is a web service that exposes simple IO functions on remote files. The plug-in performs these operations via gsoap [13]. One reason of using a web service is because of firewall issue. Direct socket connection may be more straightforward but the port for the connection is often blocked by firewalls. Each machine that contains the file must be running the web service in order to provide remote IO access. Certainly, scalability can be a problem when many replicas exist. This interface is considered as experimental since it does not address common problems on IO over wide area networks, including access control, scalability and remote IO performance. A reliable IO library that provides remote data access should be employed in a production environment. This library should, ideally, address issues such as security, portability, and performance, as in [6] and [8],

Similar to Gfarm, the Globus RLS module requires activation. When a file is being opened, the RLS plug-in firstly ensures that the module is activated, and then connects to the RLS server to get a list of physical file paths using the given logical file name, which is given in the mapping file. The list is stored in the GRF replica object. A physical file path should contain the

web service address, port, and the path of the file. The plug-in communicates with the web service and makes sure the file exists and is accessible. At the end, when requested, a network monitor is run and attached to the GRF replica object.

Below is an example of an entry in the mapping file to access replica registered in RLS:

```
myhost:input rls:rlshost:39281:data
```

Apart from the RLS server address and port, the major information is the logical file name, *data*. The plug-in uses it to retrieve a list of registered replicas from the RLS server. An example of the physical file path registered in RLS is as follows:

```
http://myhost:8080/RemoteIO:/home/tim
/input.dat
```

This tells the RLS plug-in that it needs to use the web service available on *myhost* to access the file *input.dat*.

4. Case Studies

Two case studies were conducted to illustrate the functionality and performance of GRF. For evaluation purpose, SRB, Gfarm, RLS and the Remote IO web service are all running locally. Each experiment involves running the program twenty times to get an average time. The standard deviation is also calculated.

4.1 A simple copy program

The first case study concerned a simple file copy program, built on top of GRF. This application reads data (8KB each time) from a file (140MB) and writes the data to a local file. The input file was copied to Gfarm and SRB, and was also registered in RLS. The locations of the files were stored in the mapping file. Several experiments were done for this case study and each experiment was run twenty times. During this case study, no source code modification was needed for the copy program because the file location and plug-in information are separated from the program. Only the mapping file required changes.

Plug-in	Average Time (sec)	Std. Dev.	IO Used by GRF
n/a	9.2	1.2	Unix IO without GRF
Local	9.5	1.3	Unix IO
SRB	11.9	1.1	SRB API
Gfarm	12.7	0.7	Gfarm API
RLS	15.7	1.3	Remote IO web service

Table 1 Experiment results of the file copy program

The experiment results are shown in Table 1. In the first experiment, the program was run without GRF, and conventional Unix IO was used. For the plug-in experiments, the program was run and different plug-ins were used. Not surprisingly the GRF local access plug-in has the fastest execution time comparing to the other three due to the minimal overheads within the IO. In fact, the average execution time was very similar to the conventional IO. Note that the data was stored in a local server, and thus there is no network overheads involved. Rather, the result shows that overheads in the IO layers for the replica systems would result in significantly decreased performance.

In addition, Linux commands *cp* and *scp*, as well as SRB's *Sget* command, were used to copy the same 140MB file locally in order to further compare the performance. The results are shown in Table 2. As expected, the overall performance of the plug-ins was slower than *cp* but clearly faster than using *scp*. The SRB client, *Sget*, uses multiple streams to retrieve data from the SRB server and thus it should run faster than the SRB plug-in. Yet the results show that it is not. The reason for this is unknown.

Commands	Average Time (sec)	Std. Dev.	System
cp	9.8	1	Linux
scp	19.3	0.9	Linux
Sget	14.4	1.2	SRB

Table 2 Copy 140MB using different commands

4.2 CCAM - a global climate model

The second case study concerned a global climate model, called CCAM [9]. CCAM is written in Fortran and compiled by the Intel Fortran Compiler. This model analyses the atmospheric effects over a particular area. It requires several input files during execution and writes output data to a file. Two experiments were conducted. The first experiment was to run CCAM normally without GRF for twenty times. During the second experiment, CCAM was run twenty times using GriddLeS. No code change was done in CCAM because GriddLeS allows running legacy programs on the Grid by intercepting and replacing the IO calls with the corresponding GRF IO routines.

As shown in Figure 3, some input files (such as topography and ground vegetation information) were stored in Gfarm, while the output data was written to an SRB server. Other input files exist locally, so GriddLeS and GRF were not needed. This scenario is similar to multiple organizations running applications that require shared data stored in different replica systems. With the use of GRF, the locations of the files

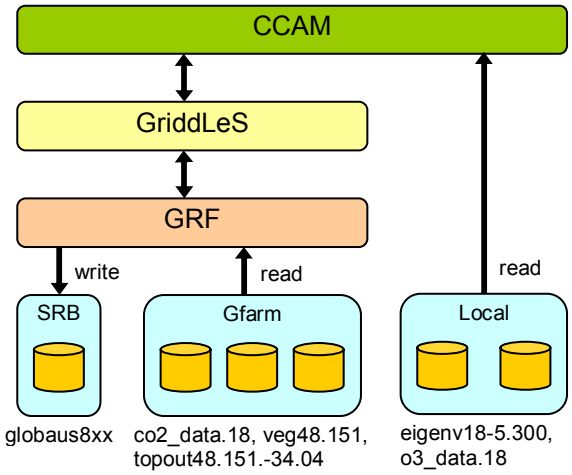


Figure 3 CCAM experiment

do not affect how the application is performed and the users are provided a flexible way to alter the file locations. Results are shown in Table 3. Average execution time increased from ~20 minutes to ~28 minutes. The execution time increased because CCAM was accessing data from different data source and various overheads exist in the IO layers.

Exp.	Average Time (min)	Standard Deviation
1	19.8	0.9
2	28.1	2.3

Table 3 Experiment results for CCAM

5. Conclusion

This paper discussed the Grid Replication Framework (GRF), which unifies the replica systems by providing an interface that abstracts the different data access mechanisms. The same set of IO routines is used to access files available in different systems. This provides applications transparent data access to the supported systems and enables easy change of data source without the need to modify the application. Importantly, building such application requires little replica system knowledge because GRF hides the complexity of the underlying IO operations. The case studies demonstrated that GRF can greatly help applications running in multiple communities where files are distributed and stored in different replica systems.

Performance improvement is performed by a data selection service. Experiments in previous studies show that optimized data access can significantly increase execution performance. Results shown in this paper further show that performing large amount of reads on small blocks of data is inefficient on the Grid because of the extra overheads. In particular, overheads within the IO layers of the different replica systems vary and can result in significant performance impact. Techniques such as pre-reading the data and buffering are needed. We plan to develop a module for GRF that provides a local data cache in order to overcome the overhead issue in the IO layers.

Reference

[1] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, B. Tierney. *Giggle: A Framework for Constructing Scalable Replica Location Services*, SC2002, Baltimore, 2002.

[2] A. Samar and H. Stockinger, *Grid Data Management Pilot (GDMP): A Tool for Wide Area Replication*, IASTED International Conference on Applied Informatics (AI2001), Innsbruck, Austria, February 2001.

[3] C. Baru, R. Moore, A. Rajasekar, M. Wan. *The SDSC Storage Resource Broker*, Proceedings of CASCON'98 Conference, Toronto, Canada, 1998.

[4] D. Abramson, J. Giddy and L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", International Parallel and Distributed Processing Symposium (IPDPS), pp 520- 528, Cancun, Mexico, May 2000.

[5] D. Abramson and J. Komineni. *Interprocess Communication in GriddLeS: Grid Enabling Legacy Software*, Technical report, School of Computer Science and Software Engineering, Monash University.

[6] I. Foster, D. Kohr, Jr., R. Krishnaiyer and J. Mogill, *Remote I/O: Fast Access to Distant Storage*, Proceedings of the Fifth Annual Workshop on I/O in Parallel and Distributed Systems, 1997.

[7] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, USA, 1999.

[8] J. B. Weissman, M. Marina and M. Gingras, *Optimizing Remote File Access for Parallel and Distributed Network Applications*, Journal of Parallel and Distributed Computing, pp 1591-1608, 2001.

[9] J. L. McGregor, K. C. Nguyen and J. J. Katzfey, "Regional Climate Simulations using a Stretched-grid Global Model", Research Activities in Atmospheric and Oceanic Modeling, H. Ritchie (ed.), pp. 3.15-3.16, 2002.

[10] L. Guy, P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger. *Replica Management in Data Grids*, Technical report, GGF5 Working Draft, July 2002.

[11] M. Manohar, A. Chervenak, B. Clifford, C. Kesselman, *A Replica Location Grid Service Implementation*, Data Area Workshop, Global Grid Forum 10, 2004.

[12] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, H. Sato, Y. Tanaka, S. Sekiguchi, Y. Watase, M. Imori, T. Kobayashi, *Worldwide Grid Data Farm for Petascale Data Intensive Computing*, Technical Report, Electro-technical Laboratory, ETL-TR2001-4, 2001.

[13] R. van Engelen and K. Gallivan, *The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks*, Proceedings of IEEE CCGrid Conference 2002.
[14] R. Wolski, N. Spring, and J. Hayes. *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*, Journal of Future Generation Computing Systems, Vol.15, No.5-6, pp.757-768, October, 1999.
[15] S. Vazhkudai, S. Tuecke, and I. Foster. *Replica Selection in the Globus Data Grid,* International Workshop on Data Models and Databases on Clusters and the Grid (DataGrid 2001), IEEE Computer Society Press, 2001.
[16] T. Ho and D. Abramson. *A GriddLeS Data Replication Service*, Proceedings of 1st International Conference on e-Science and Grid Computing, 2005.
[17] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, *Data Management and Transfer in High-Performance Computational Grid Environments*, Parallel Computing, 2001.
[18] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini. *Design of a Replica Optimisation Framework*, Technical Report, DataGrid-02-TED- 021215, CERN, Geneva, Switzerland, 2002.
[19] Y. Zhao, and Y. Hu. *GRESS - a Grid Replica Selection Service*, ISCA 16th International Conference on Parallel and Distributed Computing Systems, PDCS-2003.
[20] Data Replication Service, http://www.globus.org/toolkit/docs/4.0/techpreview/datarep/
[21] EarthScope, http://www.earthscope.org/
[22] Grid File Access Library, http://www.gridpp.ac.uk/wiki/Grid_File_Access_Library
[23] GriPhyN, http://www.griphyn.org/
[24] NVO, http://www.us-vo.org/
[25] PPDG, http://www.ppdg.net/
[26] SCEC Community Modelling Environment, http://www.scec.org/cme/
[27] Sloan Digital Sky Survey, http://www.sdss.org/
[28] The Globus Toolkit, http://www.globus.org/toolkit/
[29] The Globus Toolkit Data Management, http://www.globus.org/toolkit/data/

Active Data: Supporting the Grid Data Life Cycle

Tim Ho and David Abramson
{tim.ho, david.abramson}@infotech.monash.edu.au

Monash e-Science and Grid Engineering Lab
Faculty of Information Technology,
Monash University
900 Dandenong Road, Caulfield East, 3145, Australia

Abstract

Scientific applications often involve computation intensive workflows and may generate large amount of derived data. In this paper we consider a life cycle, which starts when the data is first generated, and tracks its progress through replication, distribution, deletion and possible re-computation. We describe the design and implementation of an infrastructure, called Active Data, which combines existing Grid middleware to support the scientific data lifecycle in a platform-neutral environment.

1. Introduction

The 21st century has been described as the century of data. Various groups predict an exponential growth in the amount of data that will be captured, generated and archived. For example, Foster predicts a ten fold increase every five years from 2000 to 2015 alone [14]. Not surprisingly, there are numerous projects addressing the computer science challenges that underpin management of this explosion in the amount of data.

Typically, data is generated in two main ways; either from real laboratory based experiments or in-silico experiments. Real experiments in areas such as high energy physics, astronomy, biology and medicine, and geophysics, capture data from scientific instruments like telescopes, micro-arrays, seismic sensors and synchrotrons. The data is usually stored in primary repositories, where additional metadata can be attached and made available for interrogation. Primary data can be replicated and distributed to a variety of geographically distributed sites, where they are processed, mined, analyzed and refined to produce secondary data sets. Primary data also comes from in-silico experiments; computer simulations that model some real world phenomenon (possibly driven by real world data) and produce output data which is then stored and analyzed in much the same way as real data. Regardless of the data source, there is a well recognized need for data curation and provenance information management [17][31]. For example, metadata such as the date and environmental conditions must be saved for a real experiment. Likewise, input parameters and date information must also be saved for an in-silico experiment. Because data from these two sources is handled in very similar ways, data management middleware typically does not distinguish between them. Thus, common solutions for handling metadata can be applied regardless of source. Tool kits such as the Storage Resource Broker (SRB) from the San Diego Supercomputing Centre [1], and the Globus RLS [22] epitomize this approach.

The problem with an exponential growth in data is that it requires an exponential growth in storage capacity. Conventionally, communities usually store and replicate everything they capture (or compute). However, this is clearly wasteful, and it is unlikely that we will be able to continue doing this for very long. There are a number of potential solutions to the storage problem. For example, compression techniques might reduce the amount of storage required. However, in some instances it may be simpler to recapture or re-compute the data, rather than to store it. This leads to the development of a *virtual data grid* [16], where data are identified by logical names and derived data from previous experiments can be re-generated when needed. Research on virtual data has attracted focused attention from a small number of groups (e.g. [9][15][21]).

One view of the re-computation problem is that we want to produce a copy of a file transparently where one does not exist. Thus, if an application opens a file that was produced by a computational model, but the data has been deleted, then the file could be re-computed without the application being aware. Clearly, implementing such a strategy requires sufficient metadata on how a data set is produced, and a set of mechanisms that can reproduce it on demand.

Further, building on the idea of a virtual data grid, we view data regeneration as a special case of

replication. When a computational model executes, it generates some output data that may later become input data of some other models. The data may be replicated to many different locations for various reasons. Later on when the data are not needed, they may be removed. In the end, this would result in complete data removal. When a reader application reads from this data, data regeneration is needed since all copies of the data are gone. We call this the *Grid Data Life Cycle (GDLC)*, shown in Figure 1, from non-existence of data, to the original copy of new data (computed), to many copies of the data (replicated), to non-existence of data (deleted), and possibly back to one copy of the data (re-computation).

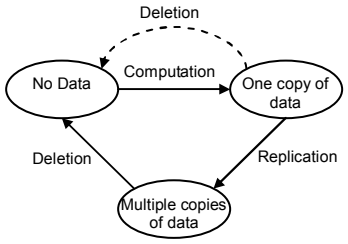


Figure 1 – Grid Data Life Cycle (GDLC)

We have developed a prototype system that manages the GDLC of the computational models and workflows in a platform neutral environment. This system, called *Active Data*, provides mechanisms to existing applications and workflows to run on the Grid. It also allows them to access data stored in different replica management systems, to associate metadata that describes how the data is computed across multiple replica systems, to ensure that data cannot be removed unless sufficient metadata for regeneration is associated, and to regenerate the required data transparently when needed during execution. Importantly, because *Active Data* is built under our GriddLeS system [6], no source code modification is needed.

2. Data Management Middleware

High performance computing applications in domains such as high energy physics, climate modelling and astronomy are often data intensive. They require access to large amounts of data, which are often copied and distributed in different locations in the Grid in order to improve reliability and system performance. While the Grid [14] describes software infrastructure that links and shares computational resources such as machines, scientific instruments and data, the term Data Grid [2] usually refers to a network of distributed storage resources that allows sharing of data distributed across remote repositories. Typically, these data can be accessed (e.g. file IO) transparently as if they were stored locally. Today, many scientific communities are using Data Grids for managing and sharing large-scale distributed data, such as [32][33][34][35][37].

Distributed data (replicas) management on Data Grids has attracted a great deal of interests (e.g. [10][16][30]). A data grid management system must tackle issues such as user authentication, access control, storage resource management, scalability, performance and data uniqueness. Further, support for metadata management is important because researchers often associate metadata with scientific datasets, and this describes the contents and structure of the associated datasets. Examples of such systems include SRB [5], Globus RLS [22] and Gfarm [23].

Developed at the San Diego Supercomputer Centre, the Storage Resource Broker (SRB) is a data grid management system that employs a client-server architecture. It supports distributed data management using global logical namespace, including data publication, replication, transfer, discovery and preservation. The SRB uses logical namespaces to identify files, storage resources, users, metadata and access controls. Because these are done independently of the physical storage systems, a logical name that maps to a file does not change even if the file is moved to a different storage system. SRB supports various file systems and provides comprehensive client commands and APIs.

When an SRB object is registered, the SRB associates system-level metadata, such as location, size and creation date, with this object. For application-level metadata, SRB users are provided comprehensive metadata support via Scommands and APIs. User-defined type of metadata can also be specified by using extensible schemas, which allows new tables to be added to the MCAT.

The Globus Toolkit [36] provides a number of components for supporting data management in the Grid. Specifically, the toolkit provides tools for data transfer (GridFTP and RFT) and replication (RLS and DRS). GridFTP [4] extends the standard FTP protocol to allow data to be transferred efficiently among remote sites. It is secure and uses multiple TCP streams for fast performance. However, one shortcoming is that when a client fails, GridFTP does not know where to restart the transfer because all the transfer information is stored in the memory. This means that a manual restart for data transfer is often required. To overcome this, the Reliable File Transfer (RFT) service [24] was developed. This non-user-based service is built on top of the GridFTP libraries and stores the transfer requests in a database rather than in memory. Clients are only required to submit a transfer request to the service and do not need to stay active because data transfer is managed by RFT on

behalf of the user. When en error is detected, RFT restarts the transfer from the last check point. The transfer status can be retrieved at anytime and RFT can also notify users when a requested transfer is complete.

Data replication is supported by the Globus Replica Location Service (RLS) [22] and, currently as a technical preview, the Data Replication Service (DRS) [3]. The RLS is an implementation of the RLS Framework [1]. It is a distributed registry that stores mapping information from unique logical identifiers to physical file locations. Using this information, RLS keeps track of the data replicas in the Grid. Client libraries and command line tools are provided for the user to register file copies, as well as to discover replica locations by querying the RLS server.

The RLS employs a metadata service, which contains information that describes the registered logical files. Descriptive data is attached to a file by creating and associating *attributes*. Currently, RLS supports values including decimal numbers, integers, strings, date and time. Attributes can be created using the provided client command. C and Java APIs are also available.

DRS is a high level data management tool that uses RLS and RFT. The main function of the DRS is to ensure files (replicas) exist on desired storage resources. This is done in three phases. The DRS firstly locates the desired files on the Grid by querying the available RLS servers. When the files, or replicas, are found, the DRS creates and submits a transfer request to RFT for copying the files to the desired storage resources. Finally, upon completion of the transfer, DRS registers the newly replicated files with the RLS servers.

Gfarm [23] also provides support for data replication and employs a metadata system to manage the file distribution. It focuses on building a distributed grid file system that provides scalable and high performance IO by integrating many local file systems and clusters. However, different to the SRB and the Globus Toolkit, the Gfarm metadata system is used to store file and parallel process information and currently does not provide support for users to associate descriptive data to the files.

The above systems all share the same goal; to provide mechanisms to manage data that are distributed geographically. However, in GDLC we also require high level tools that manage data access within workflows, data deletion when desired, and importantly, regeneration of the deleted data.

Data regeneration is a significant feature in a virtual data system, for example, as demonstrated in Chimera [15]. Chimera tracks how data is derived and allows deleted data to be recreated when needed. It uses the virtual data language (VDL) to describe procedures that generate data and records information on how these procedures may be rerun later. This information is specified as VDL definitions, which are stored in the virtual data catalog (VDC). The corresponding data definitions are used when a workflow being created requires the data. Chimera creates an abstract workflow using the data definitions and stores it as an XML document. This document is then sent to the planner that generates executable forms of a workflow. Currently, Pegasus [11] is used by Chimera as the primary planner that converts the abstract workflow to a Condor DAGman DAG, a workflow that can be executed over the Grid resources.

One limitation of Chimera is that it does not provide support for remote IO operations. It copies the whole file from remote host to the execution node after a derivation of the file is done. Whilst this is efficient when the file size is small, it is inefficient when a reader application only requires a small portion of a large dataset.

Apart from data storage and regeneration, one key aspect in the GDLC is data removal. The storage systems mentioned above all provide ways to remove data from distributed resources. If a file is no longer required, a user may decide to delete it permanently. However, when a file is used by a computational model, the user may not want to delete it unless there is a way to recreate the file. Chimera does not provide facilities for the user to remove derived data and execution will fail if no derivation definition is defined for the deleted data.

3. GriddLeS and Kepler workflows

Grid workflows refer to workflows that are executed on the Grid. Such workflows are common in many domains, such as physics, earth sciences and astronomy, and often involve intensive computation and generate large amount of derived data. These data can be in various forms, such as datasets and images, and may serve as inputs to subsequent models.

Several workflow engines exist (e.g. [13][25][29]) that allow users to compose virtual applications using existing components. In this model, each component typically takes inputs and produces outputs that would become inputs of another component. Components within a workflow usually communicate by passing files or using interprocess communication pipes. When files are used, it is often difficult to overlap processes because a reader application needs to wait until the writer application finishes writing the output file. Also, when different machines are used for each process, the output file needs to be copied. To overcome this, we developed GriddLeS [6] and have also recently integrated it to the Kepler workflow system [13].

3.1 GriddLeS

GriddLeS is a middleware layer that provides a set of interprocess communication facilities for constructing arbitrary Grid applications/workflows from existing software components [6]. Such components are connected by a mechanism called GridFiles. A GridFile is a logical file that contains user-specified information about the physical file, including, but not limited to, resource location, file path, type of storage systems and user access information. GriddLeS employs a flexible architecture for supporting different file access mechanisms, such as local file access, remote file access and replica access. Importantly, no source code modification is required to access the GriddLeS run time, and this allows legacy codes to be run using a variety of different file access mechanisms. Previous work shows that Grid workflows can easily be constructed using legacy programs (e.g. Fortran programs) and performance can be significantly improved by pipelining and overlapping workflow components in parallel processes [8][27].

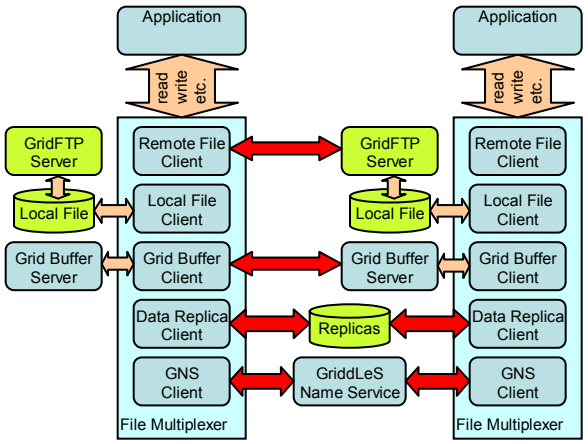


Figure 2 – The GriddLeS Architecture

GridFiles are supported by a device called the File Multiplexer (FM), which traps an application's IO system calls (e.g. open, close, read, etc.) and maps these operations dynamically to appropriate services. This IO interception mechanism enables applications to access files on the Grid without changing any code. Currently, GriddLeS provides access to local files, remote files, remote buffer for streaming, and data replicas in a number of different replica systems.

Instead of modifying the application, users need to specify the file mapping information in the GriddLeS Name Service (GNS). Each mapping in the GNS contains the original file path and the new file information. This is the central place for the users to tell GriddLeS what to do when a program is trying to access a particular file. When file path changes, a user modifies the mapping information instead of the application code.

3.2 Kepler

Recently, we have developed a number of GriddLeS actors in Kepler that allow users to specify file mapping information directly into a Kepler workflow without using the GNS [7].

Kepler is an active open source project to build and evolve a scientific workflow system [13]. This system allows scientists from multiple domains to design and execute complex scientific workflows. Kepler is built on the Ptolemy II system [26], which provides a set of Java APIs for modelling heterogeneous, concurrent and hierarchical components using various models of computations (MoCs), which govern the interactions between software components. The Ptolemy II project studies modelling, simulation and design of concurrent, real-time, embedded systems.

In Kepler, an *actor* is a computational unit. It is a reusable component in a workflow that communicates via input and output ports. The interaction styles of actors are defined by *directors*, which are responsible for implementing particular MoCs. Examples of directors are Process Network (PN) and Synchronous Data flow (SDF). The former runs actors concurrently and the latter runs actors sequentially. The workflows are controlled by the directors. When changes are required (e.g. scheduling), only the director needs to be changed and no change to the workflow graph is required.

4. Active Data

The main goal of *Active Data* (AD) is to provide an infrastructure that allows the users to manage derived data within the GDLC. Specifically, such infrastructure must provide means for the users to:

- access data stored in different storage systems;
- enhance the performance with optimized data access;
- create data replicas when needed (and access the *best* one during runtime);
- create metadata that describes how the data may be recreated;
- delete the derived data (and ensure sufficient information on data regeneration is defined);
- regenerate the data when needed; and
- avoid source code modification in existing application components.

To support these, AD integrates various existing middleware, including GriddLeS and Kepler.

4.1 Data Access

In AD, data access is supported by GriddLeS. Conventionally, the user composes workflows by specifying file mapping information in the GNS. Each component within the workflow may access local files, remote files or replicated files. Streaming between two components can also be configured by using a location-independent buffer. This means that reader and writer applications can be overlapped using a pipe in the Grid for faster performance.

Using Kepler, the workflow can also be created without using the GNS. The user simply draws a workflow graph in Kepler with the desire director (e.g. a PN or a SDF director) and the GriddLeS actors. The file mapping information is specified within each actor by connecting the output port of an actor to the input port of another actor. Users can instruct GriddLeS to copy the output files to the execution node of the next actor via GridFTP or scp. Streaming is used when PN director is in use.

Each component (i.e. actor) can be executed in different resources and user authentication is done via Globus or SSH. When PN director is used, all actors are run concurrently and a buffer will be created between each pair of actors. In this mode, the components within the workflow are connected using pipes. On the other hand, when SDF director is used, Kepler will run the actors in a sequential order and third-party file transfer will be performed at the end of each actor execution.

4.2 Data Replication and Selection

GriddLeS employs a flexible plug-in architecture for supporting different file access mechanisms for various replica management systems [28]. It provides applications transparent access to the replicated data, in which more than one copy may exist. Each supported system provides means for users to replicate data and the replica information (e.g. number of replicas and their locations, creation dates, access level and user-defined metadata) is maintained by the storage systems. Such information is useful because GriddLeS will choose the best replica during program execution based on network statistics. It can also automatically switch to a better one dynamically in runtime.

4.3 Data Deletion and Metadata

AD provides a special client command (*adrm*) for deleting replicated files. Based on the GriddLeS' unified data access framework, AD employs an abstract interface for data removal from supported systems. The users may want to specify the physical file path (i.e. replica system-specific file information) or the logical filenames that are used in the GNS. The latter is useful when the GNS is used for composing the workflows.

The major reason for this interface is to ensure that data regeneration information exists. Data can be removed permanently only when 1) there are replicas; or 2) data regeneration information associated as metadata. When the remove command is executed, it checks the existence of the file in the corresponding replica system. If there are more than one copy, it displays a list of the replicas and asks which one to be deleted. Then the command removes the chosen one permanently from the system. If the user wants to delete all replicas (or the last copy), it firstly checks whether there is data regeneration metadata associated, in which case it removes all the data but keeps the metadata in the system. An empty file still exists in the system because it is where the metadata will be associated with.

When no data regeneration metadata is available, the remove command requires the user to specify information about how data can be regenerated. Currently, AD supports regenerating data via running an arbitrary application (e.g. a linux binary), or a Kepler workflow. For an application, AD requires metadata such as physical file path to the binary, the arguments, and in the case of an application that outputs more than one file, which one is desired (Table 1). Input files for this application may be missing and require regeneration as well. If a Kepler workflow is used, then the user must save the workflow into a Kepler-specific XML document and store the document, and its path, in the replica system (Table 2).

Attribute	Value	Description
wf	no	no workflow is used
host	argos	where the program is
cmd	$HOME/ccam	the program command
arg	> output.dat	command arguments
ofile	output.dat	the output file desired

Table 1 – Metadata information for arbitrary programs

Attribute	Value	Description
wf	kepler:/argos/poll.xml	the workflow

Table 2 – Metadata information for a workflow

Clearly, a Kepler workflow is preferred over a single binary because it allows program execution on Grid resources and all execution-related information (e.g. user authentication, input/output files, program paths, resources, etc.) is stored within the workflow. With Kepler's data provenance model [12], tracking and rerunning the derivation process of the data can be performed in more complex applications. Furthermore, regeneration information is stored in the replica system

as metadata and is not specific to a particular workflow system. Hence, it is possible to support other systems, including Chimera.

One problem of this design is that not all replica systems support user-defined (application-level) metadata. For example, SRB and Globus RLS both support this but Gfarm does not. Currently, Active Data overcomes this problem by storing the information in a simple text file. This file is registered in the Gfarm system so that it is accessible via the Gfarm native interface.

4.4 Data Regeneration on demand

AD provides facilities to regenerate deleted data on demand. During runtime, when a user-specified input file is not available, AD looks for the associated metadata. If there is no metadata, the execution will be terminated because an input file is missing. When data regeneration metadata exists, AD will try to rerun the procedures that created the data previously. If a Kepler workflow is used, AD firstly retrieves the associated XML document from the replica system and uses the Kepler client command interface to execute the workflow.

The data regeneration process is transparent to the reader application. AD blocks the reader application and reruns the previous application or workflow using the information from the metadata. If the executions are on the same host, then the reader application simply reads from the local file using GriddLeS once it is created. If an execution is performed on a remote resource, then the file may either be copied, or may be accessed remotely using GriddleS' remote IO operations. Further, rather than blocking the reader application, it is possible to stream the data between the writer and reader applications using the GriddLeS Buffer Service.

5. Evaluation and Results

A few case studies have previously reported the effectiveness of the GriddLeS data access mechanisms [27][28], and demonstrated that execution performance can be improved significantly using the data source optimization. A recent case study [7] was performed using a scientific workflow that computes the air pollution scenarios from a chain of atmospheric software components. The case study shows that it is easy to compose Grid workflows in the Kepler environment using existing code. In this example, Kepler was used to create, execute and monitor the entire virtual application. The result shows that the total execution time of the five computation actors almost reduced in a half when run in parallel.

Here we conducted a case study that considers some components of the pollution model workflow, including the global climate model (CCAM) [18], the data conversion program (cc2lam), the regional weather model (DARLAM) [19], another data conversion program (lam2cit) and the photo-chemical pollution model (CIT) [20]. The main aim of this case study is to demonstrate that a deleted file can be regenerated on the fly without significant performance loss. For evaluation purpose, two experiments were performed and all data and applications were run in the same machine.

5.1 Experiment 1 – Rerun an arbitrary application

The first experiment considers a writer application (CCAM) and a reader application (cc2lam). CCAM writes climate data to the file, *globaus8xx.197901*, and this serves as input to the conversion utility, cc2lam. During the experiment, CCAM was run and the output file was stored in an SRB server. The regeneration metadata were associated with the file and this was then subsequently deleted using AD's remove command. File mapping information was entered (to the GNS). When cc2lam was executed, AD found out that the input file was removed from the SRB server. It suspended the cc2lam execution, retrieved the associated metadata and used the information to rerun CCAM. The output data of CCAM was written to the original SRB file. The experiment was repeated 10 times to get an average execution times (and standard deviations) for each model.

The results are shown in Table 3. Originally the total execution time for the two models took a bit more than 30 minutes. With the data regeneration and SRB in place, the execution took roughly 2 more minutes to complete. Such loss is expected due to the overheads in GriddLeS, and the re-computation layer.

Model	Original	Std. Dev.	Active Data	Std. Dev.
CCAM	24:43	0.9	-	-
cc2lam	6:02	1.1	-	-
Total	30:45	1.3	32.37	1.2

Table 3 – Experiment 1 results

5.2 Experiment 2 – Rerun a Kepler workflow

The second experiment concerns rerunning a Kepler workflow using the native Kepler client command. A workflow that outputs fine-grained regional weather data for an air pollution model was created. Specifically, this workflow includes these components: CCAM, cc2lam, DARLAM and lam2cit, as shown in

Figure 3. The final output file (*winds_progfc_6km.bin*) serves as the input for the CIT program. Two workflows were actually created, one using SDF director and the other one using PN director.

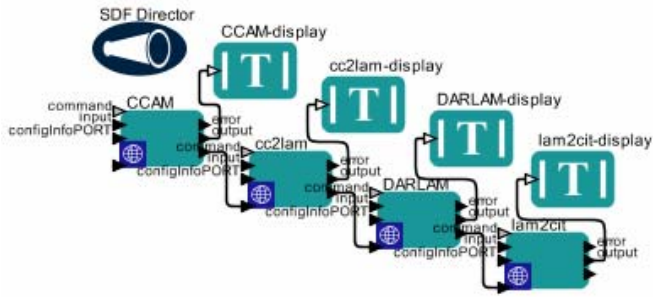


Figure 3 – Regional weather workflow using SDF

Two separate tests were performed and each test was repeated 10 times. The first test used the SDF workflow and the second one used the PN workflow. For each test, the output data of the workflow was written to the SRB server. After the execution, the workflow was saved into the Kepler XML file and this file was stored in the SRB server. The path to this workflow file was also saved as metadata in SRB. The output file was then deleted using the *adrm* command. After that, the CIT model, which requires data from the deleted file, was run.

In runtime, AD discovered that the file, *winds_progfc_6km.bin*, was missing. It then suspended CIT and re-executed the Kepler workflow using the XML document from the SRB server. Results are shown in Table 4. Rerunning the Kepler workflows took a few more minutes because of the initialization process of the Kepler environment before the workflow was run.

App.	Original (SDF)	Active Data (SDF)	Original (PN)	Active Data (PN)
Climate Model	46:01	-	39:24	-
CIT	30:24	-	14:21	-
Total	76:25	80:12	53:45	56:43

Table 4 – Experiment 2 results

6. Conclusion

This paper discusses Active Data, an infrastructure that combines existing middleware, including GriddLeS, Kepler and several replica systems, to support the GDLC. It provides support for data access, replication, selection, deletion and regeneration. Importantly, no source code modification is required and almost all existing applications can be supported.

The GriddLeS library is used extensively in Active Data because it supports legacy software components. It also employs a flexible architecture that provides applications transparent access to data in the Grid. The support for data streaming can also significantly improve the program performance. Besides, Active Data extends the virtual data concept to support data removal, where data files within workflows should not be deleted unless the system knows how to recreate them when required. Also, data generation information is stored as metadata and is not specific to any workflow systems. We plan to support more systems in the future.

Acknowledgements

The GriddLeS project is supported by the Australian Research Council and Hewlett Packard under an ARC Linkage grant, the Australian Government Department of Communications, Information Technology and the Arts under a GrangetNet grant, and the DART project. http://dart.edu.au/.

References

[1] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney, *Giggle: A Framework for Constructing Scalable Replica Location Services*, SC2002, Baltimore, 2002.

[2] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke, *The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets*, Journal of Network and Computer Applications, 23:187-200, 2001.

[3] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, and B. Moe, *Wide Area Data Replication for Scientific Collaborations*, Proceedings of 6th IEEE/ACM International Workshop on Grid Computing (Grid2005), November 2005.

[4] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, S. Tuecke, *Data Management and Transfer in High Performance Computational Grid Environments*, Parallel Computing Journal, 28:749-771, May 2002.

[5] C. Baru, R. Moore, A. Rajasekar and M. Wan, *The SDSC Storage Resource Broker*, Proceedings of CASCON'98 Conference, Toronto, Canada, 1998.

[6] D. Abramson and J. Komineni, *A Flexible IO Scheme for Grid Workflows*, IPDPS-04, Santa Fe, New Mexico, April 2004.

[7] D. Abramson, J. Kommineni and I. Altinas, *Flexible IO services in the Kepler Grid Workflow Tool*, Proceedings of 1st International Conference one-Science and Grid Computing, Melbourne, 2005.

[8] D. Abramson, J. Komineni, J. McGregor and J. Katzfey, *An Atmospheric Sciences Workflow and its*

Implementation with Web Services, Future Generation Computer Systems, 21:69-78, 2005.

[9] D. Bernholdt, S. Bharathi, D. Brown, K. Chancio, M. Chen, A. Chervenak, L. Cinquini, B. Drach, I. Foster, P. Fox, J. Garcia, C. Kesselman, R. Markel, D. Middleton, V. Nefedova, L. Pouchard, A. Shoshani, A. Sim, G. Strand and D. Williams, *The Earth System Grid: Supporting the Next Generation of Climate Modeling Research*, In Proceedings of the IEEE, 93(3):485-495, March, 2005.

[10] D. Duellmann, W. Hoschek, J. Jean-Martinez, A. Samar and B. Segal, *Models for Replica Synchronization and Consistency in Data Grid*, 10th IEEE Symposium on High Performance and Distributed Computing (HPDC-10), 2001.

[11] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi and M. Livny, *Pegasus: Mapping Scientifc Workflows onto the Grid*, Across Grids Conference, 2004.

[12] I. Altintas, O. Barney, and E. Jaeger-Frank, *Provenance Collection Support in the Kepler Scientific Workflow System*, IPAW2006, Chicago, 2006.

[13] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, *Kepler: An Extensible System for Design and Execution of Scientific Workflows*, SSDBM'04, Santorini Island, Greece, June 2004.

[14] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, USA, 1999.

[15] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao, *Chimera: A Virtual Data System for Representing, Querying and Automating Data Derivation*, In 14th Conference on Scientific and Statistical Database Management, Edinburgh, Scotland, July 2002.

[16] I. Foster, J. Vockler, M. Wilde and Y. Zhao, *The Virtual Data Grid: A New Model and Architecture for Data-Intensive Colaboration*, Proceedings of the CIDR 2003 First Biennial Conference on Innovative Data Systems Research, January 2003.

[17] J. Gray, A. S. Szalay, A. R. Thakar, C. Stoughton, and J. vandenBerg, *Online Scientific Data Curation, Publication, and Archiving*, Proceedings of SPIE, Virtual Observatories, 4846:103-107, Dec 2002.

[18] J. L. McGregor, K. C. Nguyen and J. J. Katzfey, *Regional Climate Simulations using a Stretched-grid Global Model*, Research Activities in Atmospheric and Oceanic Modeling, H. Ritchie (ed.), pp. 3.15-3.16, 2002.

[19] J. L. McGregor, K. J. Walsh and J. J. Katzfey, *Nested Modelling for Regional Climate Studies*, A. J. Jakeman, M. B. Beck and M. J. McAleer (eds.), Modelling Change in Environment Systems, J. Wiley and Sons, pp. 367-386, 1993.

[20] K. J. Tory, M. E. Cope, G. D. Hess, S. H. Lee and N. Wong, *The use of long-range transport simulations to verify the Australian Air Quality Forecasting System*, Australian Meteorological Magazine, 52(4):229-240, 2002.

[21] L. Bright and D. Maier, *Deriving and Managing Data Products in an Environmental Observation and Forecasting System*, In Proceedings of Conference on Innovative Data Systems Research (CIDR), 2005.

[22] M. Manohar, A. Chervenak, B. Clifford and C. Kesselman, *A Replica Location Grid Service Implementation*, Data Area Workshop, Global Grid Forum 10, 2004.

[23] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, H. Sato, Y. Tanaka, S. Sekiguchi, Y. Watase, M. Imori, and T. Kobayashi, *Worldwide Grid Data Farm for Petascale Data Intensive Computing*, Technical Report, Electro-technical Laboratory, ETL-TR2001-4, 2001.

[24] R. Madduri, C. Hood, W. Allcock, *Reliable File Transfers in Grid Environments*, Proceedings of the the 27th IEEE Conference on Local Computer Networks, 2002.

[25] S. Majithia, M. S. Shields, I. J. Taylor, and I. Wang, *Triana: A Graphical Web Service Composition and Execution Toolkit*, Proceedings of the IEEE International Conference on Web Services (ICWS'04), 2004.

[26] S. S. Bhattacharyya, E. Cheong, J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, and H. Zheng, *Heterogeneous Concurrent Modeling and Design in Java*, Technical Memorandum UCB/ERL M02/23, EECS, University of California, Berkeley, August 5, 2002.

[27] T. Ho and D. Abramson, *A GriddLeS Data Replication Service*, Proceedings of 1st International Conference one-Science and Grid Computing, Melbourne, 2005.

[28] T. Ho and D. Abramson, *A Unified Data Grid Replication Framework*, Proceedings of 2st International Conference one-Science and Grid Computing, Amsterdam, 2006.

[29] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat and P. Li. *Taverna: A tool for the composition and enactment of bioinformatics workflows*, Bioinformatics Journal, 20(17):3045-3054, 2004.

[30] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, C. Salisbury and S. Tuecke, *The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets*, Journal of Network and Computer Applications, 23:187–200, 2001.

[31] Y. L. Simmhan, B. Plale, D. Gannon, *A Survey of Data Provenance in e-Science*, SIGMOD Record, 34(3):31-36, 2005.

[32] BIRN: Biomedical Informatics Research Network. http://www.nbirn.net.

[33] National Virtual Observatory. http://www.srl.caltech.edu/nvo/.

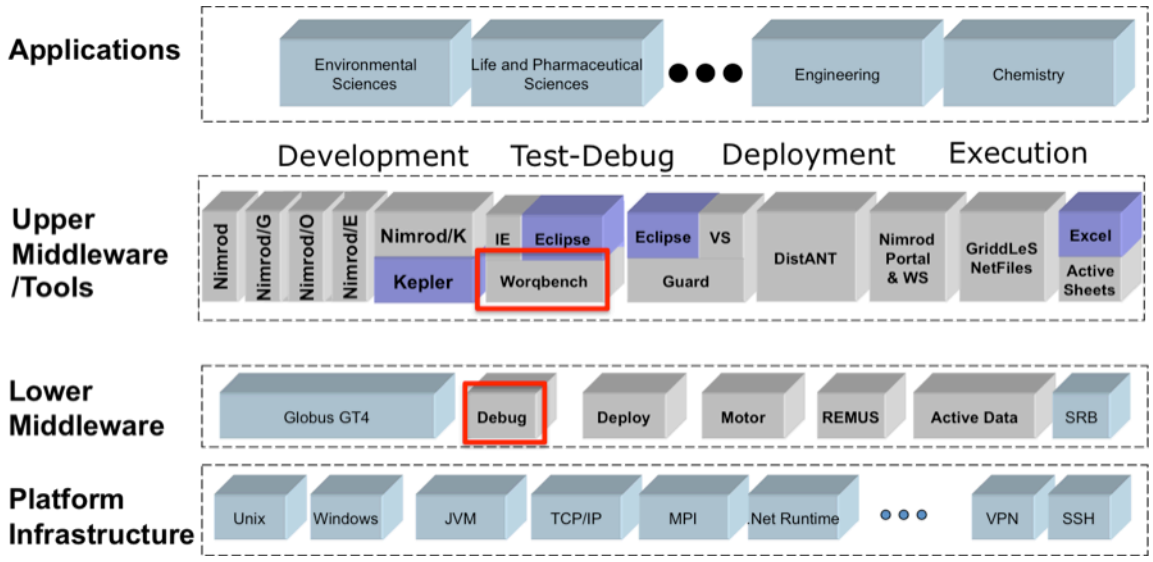
[34] Network for Earthquake Engineering Simulation. http://www.eng.nsf.gov/nees/.

[35] ROADNet: Real-time Observatories, Applications and Data Management. http://roadnet.ucsd.edu.

[36] The Globus Toolkit. http://www.globus.org/toolkit/.

[37] The Grid Physics Network. http://www.griphyn.org/proj-desc1.0.html

Chapter 12: Grid Debugging



Relative debugging as discussed in Chapter 10 addresses a niche problem in testing. In this chapter we discuss more general tools for performing debugging across the distributed Grid infrastructure. (53) introduces the idea of a Grid debugger, and (54) discusses its implementation in an integrated development environment. (55) discusses an overall IDE enabled framework for application development.

(1) Kurniawan, D. and Abramson, D. "A WSRF-Compliant Debugger for Grid Applications", 21st IEEE International Parallel & Distributed Processing Symposium (IPDPS 2007) March 26-30, 2007, Long Beach, California USA. 12-1 .. 12-10

(2) Kurniawan, D and Abramson, D, "An Integrated Grid Development Environment in Eclipse", The 3rd IEEE International Conference on e-Science and Grid Computing, Bangalore, India, IEEE Computer Society, Los Alamitos CA USA, pp. 491-498. 10-13 December 2007. 12-11 .. 12-18

(3) Kurniawan, D and Abramson, D. "An IDE Framework for Grid Application Development", Grid 2008, Sept 29th – Oct 1st, Tsukuba, Japan. 12-19 .. 12-26

Contributions

At the time this work was performed, Kurniawan was a PhD student under my supervision.

A WSRF-Compliant Debugger for Grid Applications

Donny Kurniawan and David Abramson
Monash e-Science and Grid Engineering Lab
Faculty of Information Technology, Monash University
{donny.kurniawan, david.abramson}@infotech.monash.edu.au

Abstract

Grid computing allows the utilization of vast computational resources for solving complex scientific and engineering problems. However, development tools for Grid applications are not as mature as their traditional counterparts, especially in the area of debugging and testing. Debugging Grid applications typically requires a programmer to address non-trivial issues such as heterogeneity, job scheduling, hierarchical resources, and security. This paper presents the design and implementation of a Grid service debug architecture that is compliant with the Web Service Resource Framework standard. The debugger provides a library with a set of well-defined debug APIs.

1. Introduction

Grid computing facilitates the aggregation of geographically distributed resources such as data servers, compute clusters, and scientific instruments to act as a single large system. This virtualisation allows multiple organizations to cooperate on solving large-scale problems which cannot be addressed by a single organization [1]. Grid middleware provides a set of services for managing and controlling the resources and the infrastructure. The services include security, resource management, data management, and information services. This middleware defines protocols and standards for users to manage program execution, to access data, to monitor resources, and to query information [14].

Applications need to be specifically written to take full advantage of the remote resources. However, currently Grid programmers still rely on traditional tools and techniques that are designed for local development. For example, applications are typically written on a programmer's desktop and are transferred to remote resources to be run. Debugging these applications requires the programmer to log in to the remote nodes and to run a local debugger [2]. These development techniques are time consuming and error prone and we believe will hinder the adoption of the Grid.

In the case of Grid application debugging, the distributed and heterogeneous nature of Grids presents a challenge specifically in the areas of heterogeneity, job scheduling, resource hierarchy, and security.

Heterogeneity: A Grid testbed may consist of resources with different computer architectures, operating systems, and debuggers. Some of these differences are addressed by current Grid middleware such as Globus [10], for example, by providing uniform ways of invoking applications and transferring data files. However, these do not support debugging, and thus the underlying heterogeneity is still exposed to the programmer.

Job scheduling: In a typical Grid environment program execution is managed by a local queue manager, which schedules jobs according to criteria such as resource availability and processor load. This batch processing scenario makes it difficult to debug an application interactively because the programmer cannot easily determine when the job actually starts execution. As a result, programmers must resort to ad-hoc techniques such as polling the scheduler regularly to check whether the application has started, and then they must manually attach a debugger to the process. In a Grid, this process might need to be repeated across multiple resources, possibly using different local schedulers, making the technique cumbersome and error prone.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

Hierarchical resources: Many Grid testbeds are actually built from distributed clusters, consisting of a single front-end machine and multiple back-end processor nodes. These back-end nodes are usually only accessible from the front-end, which may in turn be behind a gateway server or a firewall. Thus, debugging an application running on the execution nodes of a cluster may require access to a hierarchy of intermediate computers. However, depending on the Grid security policy in place, a programmer may not have direct access to all of these machines, again complicating debugging.

Security: Grid level debugging must be secure. For example, a debugger run by one user must be restricted from attaching to and controlling another user's processes. Clearly a debugger must conform to the security framework implemented in the Grid middleware. Thus, if the job invocation mechanism uses X.509 certificates, such as used by Globus, then the debugger must also operate in this framework.

We believe that many of these issues can be solved by building the debug architecture into the Grid fabric. This research paper focuses on the specification and design of a standard set of application programming interfaces (APIs) suitable for debugging and testing computational Grid applications. These services can then be used within a Grid level debugger, or other high level software tools that require these functions. The architecture is modular and independent of the particular Grid middleware and back-end debug servers that are used.

The rest of the paper is organized as follows. Section 2 presents the design and architecture of the debugger. The implementation details are described in section 3. In section 4, we outline related work in the area of Grid debugging tools. The discussion on how our approach meets the challenges is given in section 5. Section 6 presents the conclusion.

2. Design and architecture

The debugger is not designed as a stand-alone monolithic program but rather as component-based software which includes a plug-in or an extension to Grid middleware. This design simplifies adding a debugging service to existing middleware. The debugger is composed of four components: the client, the middleware compatibility layer, the debug library, and the debug back-end. The components and the overall architecture are shown in figure 1.

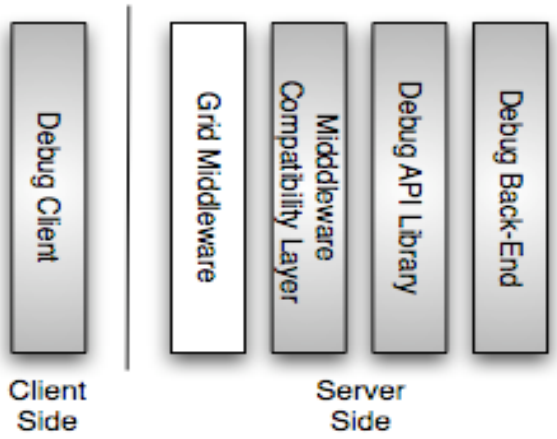


Figure 1. Debugger architecture

Debug client. A developer utilizes the debugging service by using a debug client that is written according to client specification of the Grid middleware. It takes advantage of various services offered by the middleware such as security and notification services. Various APIs in the debug library are utilized by the client, which can then be implemented as a simple command-line interface (CLI) program similar to a traditional CLI debugger or as a plug-in of an IDE such as Eclipse or NetBeans.

Middleware compatibility layer. The compatibility layer is a Grid middleware plug-in that wraps the debug library as a Grid debugging service. The layer is written according to the extension mechanism of the middleware. This component acts as a translation layer between middleware-specific and generic debug interface.

Debug library. The main component of the system is a debug library with a set of well-defined debug APIs based on the High Performance Debugging Forum (HPDF) standard [9]. HPDF was chosen because it is the result of significant research on an appropriate command set for a parallel debugger, and it serves as a sound base for a debug library interface. The API defines a generic debug model and a collection of methods and objects for Grid application debugging. The debug library is middleware-

independent and is linked to the compatibility layer.

HPDF is a collaborative effort in the area of parallel and distributed systems with a goal to define standards for debugging tools for high-performance computers. HPDF has defined a standard for command-line interface parallel debuggers. This standard forms the model of our debug API. The API specifies objects and methods that correspond to commands in the HPDF standard. Not all commands are translated into methods since some of them are only peculiar to CLI tools. The API includes, for example, methods to define process sets, to create/delete breakpoints, and to control the execution of programs. The API is discussed in more detail in section 2.1.

Debug back-end. The debug back-end provides functionalities as defined by the API. It is the component that performs the actual low-level debug operations, for example ptrace system calls, on the debugged application. A traditional debugger such as GDB can be utilized as the back-end debugger by implementing an interface to the debug library.

2.1. Debug API

The debug API specifies a debug model, methods, and data structures using an object-oriented paradigm. Classes are defined to represent entities such as processes, breakpoints, and events. Although the definitions are language-independent, we adopt Java programming language to implement the debug library. The design of the API is based on the HPDF standard. HPDF commands, for example, focus, defset, load, and step are implemented as methods (see [15] pages 20 and 21 for definitions of these). However, there are several key differences between the API and the standard which are discussed below.

CLI-specific features. The HPDF standard also defines CLI-specific features such as command history and debugger state variables. These features relate to the user interface aspect of CLI tools and they are not pertinent to API-based debuggers. The debug API does not keep track of command history, however, such functions can be implemented in the debug client for user convenience. A class, DebugConfig, is provided to store configuration parameters. Various DebugConfig methods can be called to change the behaviour of the API.

Output and events. In response to user input, a debugger typically issues a variety of messages through an output stream to a terminal or to a GUI window. Instead of a stream, the debug API employs an event-based mechanism with a set of methods and a queue. The debug client checks the event queue at regular intervals to retrieve debug messages and output. Alternatively, if the Grid middleware supports a notification service, the client can utilize it to be notified of any messages from the debug library.

Processes and threads. HPDF recognizes three models of parallelism: processes-only, threads-only, and multilevel (multi-process and multi-thread). However, the current implementation of the debug library only supports multi-process debugging. Various methods, for example, focus, defSet, and undefSet are provided that allow programmers to debug multiple processes easily.

Remote debugging. The debug API is augmented with a remote debugging feature. Although not specified in the HPDF standard, the feature gives more flexibility in Grid application debugging. The extension allows a remote debugger such as GDB/GDBServer to be utilized as the back-end debug engine.

Debug sessions. The debug API is accessed through a DebugSession object which represents a single debugging session for a user (figure 2). It comprises objects for event notification (DebugEventManager), for storing configuration parameters (DebugConfig), for remote debugging (RemoteDebugManager), and for accessing the underlying back-end debugger (IDebugger). The API does not include any security-related functionality such as user identification and authorization. Instead, it relies on the security service provided by the Grid middleware. Table 1 lists the currently implemented interfaces.

```
General Debugger Interface
DebugVariable[] set()
DebugVariable set(String var)
void set(String name, String val)
void unset(String var)
void unsetAll()
```

```
Process Sets
void focus(String name)
void defSet(String name, int[] procs)
void undefSet(String name)
void undefSetAll()
DebugProcess[] viewSet(String name)

Debugger Initialization/Termination
void load(String prg)
void load(String prg, int numProcs)
void run(String[] args)
void run()
void detach()
void exit()

Program Information
DebugStackFrame[] where()

Data Display and Manipulation
String print(String expr)

Execution Control
void step()
void stepSet(String set)
void step(int count)
void stepSet(String set, int count)
void stepOver()
void stepOverSet(String set)
void stepOver(int count)
void stepOverSet(String set, int
count)
void stepFinish()
void stepFinishSet(String set)
void halt()
void haltSet(String set)
void cont()
void contSet(String set)

Actionpoints
void breakpoint(String loc)
void breakpointSet(String set, String
loc)
void breakpoint(String loc, int
count)
void breakpointSet(String set, String
loc, int count)
void breakpoint(String loc, String
cond)
void breakpointSet(String set, String
loc, String cond)
void watchpoint(String var)
void watchpointSet(String set, String
var)
DebugActionpoint[] actions()
DebugActionpoint[] actions(int[] ids)
DebugActionpoint[] actions(String
type)
void delete(int[] ids)
void delete(String type)
void disable(int[] ids)
void disable(String type)
void enable(int[] ids)
void enable(String type)

Miscellaneous
void remote(String host, int port)
void remoteManagerServe(int infPort,
int supPort)
void remoteManagerDestroy()
boolean
remoteManagerIsInferiorConnected()
boolean
remoteManagerIsSuperiorConnected()
DebugEventQueue getEventQueue()
```

Table 1. Debug API

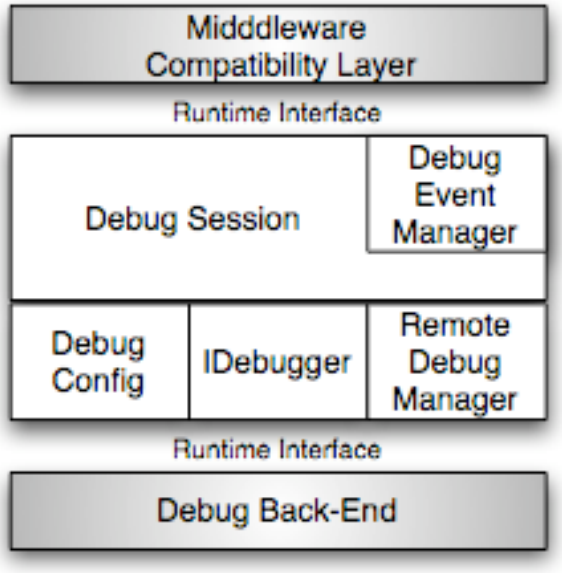


Figure 2. Debug library components

3. Implementation of the debugging service

We have implemented a Grid debugging service using two technologies: Globus Toolkit 4 as the Grid middleware layer and GDB [4] as the back-end debug engine. Sections 3.1 and 3.2 give more descriptions about GT4 and GDB. Section 3.3 presents a detailed explanation and outlines a sequence of events that happen in a debugging session. Section 3.4 discusses two clients of the debugging service.

3.1. Globus Toolkit 4 and WSRF

Globus Toolkit has been developed since the late 1990s to support the development of service-oriented distributed applications and infrastructures [10]. It enables easy federation of distributed resources such as data storage, compute clusters, networks, and remote sensors. The Globus Toolkit is currently the predominant middleware deployed on Grid resources.

The latest release of the toolkit, Globus Toolkit 4 (GT4), employs extensive use of Web Services to define its interfaces and component structures. Web Services are of special interest since they are implementation and platform independent, and their role of interconnecting various systems is similar to the role of Grid computing [11]. However, traditional Web Services are stateless which prevents them from retaining data between invocations. Various workarounds exist such as browser cookies and session identifications to enable stateful Web Services. Nevertheless, these workarounds are non-standard and may not allow communication between services.

WSRF (Web Service Resource Framework) is a set of proposed specifications that provides a standard-defined way to create stateful Web Services [12]. These OASIS-published specifications allow Web Services to retain their states while communicating with each other or with other resources. Globus Toolkit 4 has been designed and implemented around WSRF. Its services are WSRF-compliant and provide API with C and Java bindings.

On GT4 resources, the actual execution of Grid applications is handled by WS-GRAM. WS-GRAM (Web Services Grid Resource Allocation and Management) is the execution manager of Globus Toolkit. It is a set of WSRF-compliant Web Services that allows users to submit, to monitor, and to cancel jobs on Grid resources. WS-GRAM itself is not a job scheduler. It utilizes the standard fork() system call or a local job scheduler, for example, PBS or LSF on the resources, but importantly, these are virtualised and are not visible to the client processes.

3.2. GDB and GDBServer

The GNU Debugger (GDB) [4][13] is the de facto source-level debugger used on many computer architectures. It supports programming languages such as C, C++, and Fortran. GDB can act as a software-controllable back-end debugger through the use of GDB/MI which is a machine oriented text interface to GDB. The interface provides means for a high-level debugger to be built on top of GDB.

An advantage of using GDB as the back-end debug engine is the remote debugging feature which is handled by an auxiliary program called GDBServer. It performs ptrace operations on a debugged program on remote machines. Figure 3 shows the interaction between GDB and GDBServer and how they work:

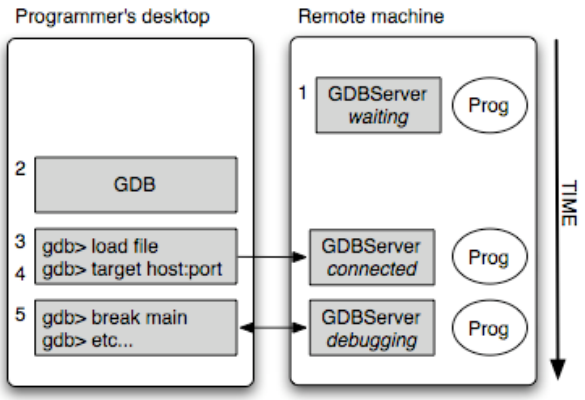


Figure 3. GDB and GDBServer interaction

1. A user logs in to a remote machine and invokes GDBServer with a program to be debugged. GDBServer then waits for a connection from GDB.
2. GDB is started on a programmer's desktop.
3. The debugging symbol file of the program is loaded in GDB.
4. The user instructs GDB to connect to the remote target using a TCP/IP connection. The user specifies the host name and the port number.
5. The user debugs the remote program as usual with GDB as if the debugged program runs on the programmer's desktop.

Remote debugging using GDBServer has some advantages which are listed below:

- The size of GDBServer is very small and it does not overload remote nodes.
- The debugging symbol resolution is performed at the GDB end thus a program on a remote machine does not need to be compiled with symbols included. However, the user must have another copy of the program with the debugging symbols on the local machine.
- GDB and GDBServer support debugging of cross-compiled applications. This allows the user to debug a program on a remote machine which has a different computer architecture from the user's desktop.

Nevertheless, in the Grid environment, GDBServer is not suitable for remote debugging since it does not address issues such as security, job scheduling and hierarchical resources. To solve this problem, we propose implementing a callback notification in GDBServer. We reversed the waiting-connecting mechanism in GDB and GDBServer. In this mechanism, it is GDB that waits for a callback connection from GDBServer. This technique alleviates the need to constantly poll the scheduler to check whether an application has been started. In a testbed with a hierarchy of resources, a programmer does not need to have access to intermediate machines, provided that the back-end node where GDBServer is running can access the programmer's desktop. This technique also adds another layer of security since the debugging activity is initiated by the application rather than by the programmer thus a user cannot debug another user's processes. A sample debugging session to illustrate this is given in section 3.3.

3.3. Debugging a Grid application

An example of debugging a Grid application is shown in figure 4. The description of the sequence of events that happen in the debugging session:

1. A Grid user submits a job to WS-GRAM. The submitted job includes GDBServer and the application. Host address and port number details for GDBServer are also given as arguments. The user must have a valid Globus certificate to access WS-GRAM.
2. WS-GRAM passes the job to a job scheduler. Depending on the scheduling criteria, the job may not be executed immediately.
3. Independently, the user invokes a debug client that interacts with the Globus compatibility layer called WS-DBG. WS-DBG acts as a wrapper for the debug library and exposes the library as a WSRF-compliant debugging service. The library then starts GDB (arrow 6) that waits for a connection from GDBServer. The same X.509 certificate that is used to access WS-GRAM is also required to access WS-DBG.
4. At a scheduled time, the job scheduler executes the job. GDBServer is started to debug the application. Details of the host address and port number are passed to GDBServer.
5. GDBServer contacts GDB using a TCP/IP connection and establishes a two-way communication (arrow 7).

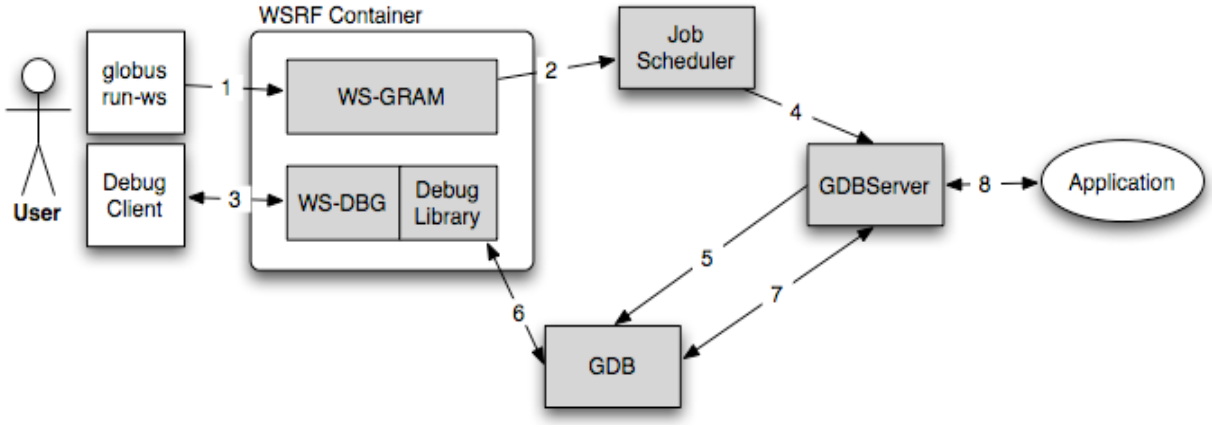


Figure 4. Debugging a Grid application

6. The debug library communicates with GDB. API methods invoked by the user are translated as GDB commands using the GDB/MI interface. GDB instructs GDBServer (arrow 7) to perform the corresponding ptrace operations (arrow 8).

3.4. Debug client

We developed a simple command-line interface debug client to test the implementation of the debugger. It simply accepts inputs from a user and sends them to the debug service. The client is linked with GT4 client-side libraries and utilizes the security and authentication services of the middleware. A sample debugging session is given below.

```
donny@attica:~$ grid-proxy-info
...
path     : /tmp/x509up_u1000
timeleft : 99:19:29  (4.1 days)

donny@attica:~$ java org.globus.
monash.clients.DebugService.DebugClie
nt https://130.194.224.235:8443/wsrf/
services/monash/core/DebugFactoryServ
ice
Acme Debugger:
::> load /tmp/hello
::> waitForExecution
Inferior not connected yet
Inferior not connected yet
...
Inferior not connected yet
Ready to debug...
::> run
Debugger Out: Running
Program Out: Hello World
Program Out: Variable = 10
::> exit
Exiting...
donny@attica:~$
```

To use the Globus debug service, a user must have a valid Globus credential which is shown here by the grid-proxy-info command. The user loads the symbol file of a program and waits for a notification from the debug service. In another terminal, the user submits a job to WS-GRAM using the globusrun-ws command. The debug client is notified when the program is ready to be debugged (indicated by the "Ready to debug" message).

In addition, using a program such as Jython that integrates Python with Java, the debug service can be accessed by clients written in other languages as demonstrated below.

```
donny@attica:~$ jython
Jython 2.1 on java1.4.2-02 (JIT:
null)
Type "copyright", "credits" or
"license" for more information.
>>> import PyDebugClient
:: successfully loaded
>>> PyDebugClient.debug.debugInit
(PyDebugClient.DebugInit())
org.globus.monash.stubs.DebugService_
instance.DebugInitResponse@1
>>> PyDebugClient.debug.debugLoad
("/tmp/hello")
org.globus.monash.stubs.DebugService_
instance.DebugLoadResponse@1
>>> PyDebugClient.debug.debugBreak
point("main")
org.globus.monash.stubs.DebugService_
instance.DebugBreakpointResponse@1
>>> PyDebugClient.debug.debugRun
(PyDebugClient.DebugRun())
org.globus.monash.stubs.DebugService_
instance.DebugRunResponse@1
>>> PyDebugClient.debug.debugStepOver
(PyDebugClient.DebugStepOver())
org.globus.monash.stubs.DebugService_
instance.DebugStepOverResponse@1
>>> PyDebugClient.debug.debugPrint
("var")
'10'
>>>
```

4. Related work

There are several debuggers that are designed for testing and debugging Grid applications. Some examples of them are: p2d2 [3], the metadebugger in the Harness framework [5], Net-dbx-G [6], the Mercury Monitoring System [7], and PDB [8].

The Portable Parallel/Distributed Debugger (p2d2) is a project at the NASA Ames Research Center that developed a debugger for applications running on heterogeneous computational Grids [3]. It employs a client-server architecture and relies on GDB [4] as the low-level portable debugger. Instances of GDB communicate with a debug server which is implemented in C++ and maintains a collection of C++ objects to represent processes and stacks. The server is controlled by a

graphical user interface, the debug client, that allows users to examine the state and to control the execution of Grid applications. The current implementation of p2d2 only supports Globus Toolkit 2 jobs.

Harness is a metacomputing system that defines a simple but powerful architectural model to overcome the limited flexibility of traditional distributed software frameworks [5]. It consists of a kernel and plug-ins that provide various services for users. It is implemented in Java to leverage the homogenous architecture, the JVM, over heterogeneous computer platforms. Harness provides a distributed virtual machine for execution of metacomputing applications written in Java with Remote Method Invocations (RMI). A metadebugger has been developed in the Harness framework using the Java Platform Debug Architecture with remote debugging capability. The debugger is closely intertwined with the framework and it cannot be used with other Grid middleware.

Net-dbx-G is a web-based debugger for Message Passing Interface (MPI) programs executing on Grid resources [6]. It uses Java applets as the user interface and GDB [4] as the back-end debugger. It supports Globus Toolkit 2 (GT2) middleware with MPICH-G2 that provides the MPI programming library. Debugging an application requires the executable to be compiled with the Net-dbx-G instrumentation library. When the application is executed, an initialization method in the library notifies the debug client and spawns a child GDB process to debug the parent process. The user then controls GDB from a web browser using the Java applets.

The Mercury Monitoring System which is developed as part of the GridLab project is a generic Grid monitoring framework [7]. It provides support mainly for application monitoring, however, the recent release of Mercury allows users to perform remote debugging. It employs GDB and GDBServer [4]. GDBServer is used as the debug server on Grid nodes with GDB as the user interface on a programmer's desktop. Debugging is performed by sending a message to the Mercury monitoring library that is compiled into the application. The library then forks GDBServer and instructs it to attach to the debugged process. The developer controls the debug server using GDB from the desktop.

PDB is an implementation of the pervasive debugging approach [8]. It leverages the Xen Virtual Machine Monitor to virtualise the system resources used by a debugged application. The virtualisation allows a user to control and to inspect the complete state of the application and the resources including their low-level details such as processor instructions, system timers, and thread schedulers. By using Xen to virtualise Grid resources, users can deterministically debug Grid applications. However, this deterministic debugging technique is difficult to attain for applications that need to be run on a large-scale distributed system.

Different to all the debuggers described, our debugger is designed as a debug library with high-level application programming interfaces (API) suitable for debugging Grid applications. It has a layered and modular architecture that allows it to be plugged into any Grid middleware and to be used with any debug back-ends. Our approach simplifies the development process by not requiring the debugged applications to be compiled with an instrumentation/monitoring library.

5. Meeting the challenges

Rather than developing, yet again, a debugger for a particular Grid middleware, we have designed and implemented a modular library that can accommodate different Grid middleware and different debug back-ends. This design simplifies extending existing middleware with a debugging service and leverages existing debuggers. Furthermore, the library can be utilized by other tools. For example, profilers and high level tracers can be built on top the library. In addition, tools for enforcing software contracts (pre and post conditions) [16] for Grid applications could be implemented with ease by leveraging the APIs.

The current implementation of a Grid debugging service utilizes Globus Toolkit 4 as the middleware layer and GDB as the debug engine. A discussion on how this service meets the challenges listed in section 1 is given below.

Job scheduling. By implementing a callback mechanism in the service, the task of initiating a

debugging session is inherent in the way an application is initiated. This mechanism alleviates the need for a programmer to constantly poll the job scheduler to check whether the application has been started.

Hierarchical resources. The callback mechanism eliminates the need for the programmer to access intermediate resources (firewall server, gateway machine, or front node) to debug an application. The programmer is required, however, to wait for an incoming connection from the execution node where the debugged application is running and this is managed by the new framework.

Security. Our debugging service is implemented as a library that leverages the security and authentication services provided by Grid middleware. It does not require changes to the Grid security policy in place. In addition, since the debugging activity is initiated by the application rather than by the programmer, a user cannot arbitrarily debug another user's processes.

Heterogeneity. The issue of heterogeneity is addressed by proposing and implementing standard methods and API that could support a number of tools and middleware for Grid application debugging.

A debugger for programs running on heterogeneous architectures must support various data representations that occur because of different architectural features such as byte ordering and word length. A significant body of research has been conducted by our group on an architecture independent data format (AIF) [19][20][21]. When data is converted to AIF, it is labeled with a format descriptor string that describes the layout and the size of the data. AIF facilitates means for addressing machine heterogeneity in debugging Grid applications.

We plan to adopt the Grid debugging service described in this paper as part of our integrated framework for Grid application development [17]. In addition, because the library is modular, it could be adopted as the underlying debugging service for other Grid tools such as g-Eclipse [18].

6. Conclusion

This research paper presents the design of a Grid debugging service and an implementation of the service in the form of a WSRF-compliant debugger for Grid applications. A debug library with a set of well-defined debug API based on the High Performance Debugging Forum (HPDF) standard is also described. While a complete service using Globus Toolkit 4 and GDB has been implemented, further testing and implementation using other Grid middleware and debug back-ends need to be conducted.

7. References

[1] Foster, I., Kesselman, C. and Tuecke, S. (2001) The anatomy of the grid: enabling scalable virtual organizations. International Journal of Supercomputer Applications, 15, 200-222.

[2] Balle, S. M. and Hood, R. T., GGF UPDT User Development Tools Survey, March 2004.

[3] Hood, R. and Jost, G. A Debugger for Computational Grid Applications. Heterogeneous Computing Workshop 2000: 262-270.

[4] GDB: The GNU Debugger Home Page, http://www.gnu.org/software/gdb/gdb.html.

[5] Lovas, R. and Sunderam, V. Debugging of Metacomputing Applications.
Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS-JPDC), Fort Lauderdale, FL, USA, 2002.

[6] Neophytou, P., Neophytou, N., Evripidou, P. Net-dbx-g: A Web-based Debugger of MPI Programs Over Grid Environments, Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium, 19-22 April 2004, Pages:35 - 42.

[7] Gombas, G., Marosi, C. A., and Balaton, Z. Grid Application Monitoring and Debugging Using the Mercury Monitoring System. EGC 2005, European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005: 193-199.

[8] Mehmood, R., Crowcroft, J., Hand, S., and Smith, S. Grid-Level Computing Needs Pervasive Debugging. Proceedings of Grid 2005, 6th IEEE/ACM International Workshop on Grid Computing Seattle, Washington, USA, November 2005.

[9] High Performance Debugging Forum's HPD Version 1 Standard: Command Interface for Parallel Debuggers, (Rev. 2.1), 1998. http://www.ptools.org/hpdf/draft.

[10] Foster, I. Globus Toolkit Version 4: Software for Service-Oriented Systems. IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2005.
[11] Atkinson, M. Rationale for choosing the Open Grid Services Architecture, in Grid Computing: Making the Global Infrastructure a Reality, Wiley, 2003.
[12] The WS-Resource Framework, http://www.globus.org/wsrf/specs/ws-wsrf.pdf.
[13] Debugging with GDB, http://sources.redhat.com/gdb/current/onlinedocs/gdb.html.
[14] Joseph, J. and Fellenstein, C., Grid Computing, Prentice Hall, 2004.
[15] Francioni, J and Pancake, C., High Performance Debugging Standards Effort, http://web.engr.oregonstate.edu/~pancake/papers/HPDebugForum.pdf
[16] Meyer, B., Design by Contract, in Advances in Object-Oriented Software Engineering, eds. D. Mandrioli and B. Meyer, Prentice Hall, 1991, pp. 1-50
[17] Kurniawan, D., Abramson, D., Worqbench: an Integrated Framework for e-Science Application Development. e-Science meeting 2006, Netherlands, December 2006.
[18] g-Eclipse – Access the Grid, http://www.geclipse.org.
[19] Abramson, D.A., Sosic, R. and Watson, G., Implementation Techniques for a Parallel Relative Debugger, International Conference on Parallel Architectures and Compilation Techniques - PACT '96, October 20-23, 1996, Boston, Massachusetts, USA.
[20] Watson, G. and Abramson, D., Relative Debugging For Data Parallel Programs: A ZPL Case Study, IEEE Concurrency, Vol 8, No 4, October 2000, pp 42 – 52.
[21] Abramson, D., Finkel, R., Kurniawan, D., Kowalenko, V. and Watson, G., Parallel Relative Debugging with Dynamic Data Structures, 16th International Conference on Parallel and Distributed Computing Systems, August 13 - 15, 2003 Reno, Nevada, USA.

An Integrated Grid Development Environment in Eclipse

Donny Kurniawan and David Abramson
Monash e-Science and Grid Engineering Lab
Faculty of Information Technology, Monash University
{donny.kurniawan, david.abramson}@infotech.monash.edu.au

Abstract

With the proliferation of Grid computing, a large number of computational resources are available for solving complex scientific and engineering problems. Nevertheless, it is non-trivial to write, deploy, and test Grid applications over heterogeneous and distributed resources. Further complicating matters, programmers may need to manually manage variations in source code due to resource heterogeneity. This paper presents an implementation of an integrated Grid development environment that leverages IBM's Eclipse IDE and our application development framework, Worqbench. It provides novel tools to develop and debug Grid software. It regards resources as first-class objects in the IDE and allows tight integration between the test beds and the code development process. We discuss how the environment assists programmers in developing Grid applications.

1. Introduction

Grid computing allows widely distributed resources such as scientific instruments, clusters, and databases to be tied together to give the appearance of a single large system [1]. It enables e-Science and future research on large-scale problems, which require high-performance computing and terabytes of data storage. This requirement may only be addressed collectively by several organizations. Grid computing requires particular software, Grid applications to take advantage of the underlying architecture and resources. A Grid test bed may consist of heterogeneous resources with different architectures and operating systems and this variability must be taken into account by the applications. Apart from heterogeneity, additional considerations also include issues such as security, resource management, information services, and data management [2]. To mitigate some of these concerns, Grid middleware has been developed. This middleware provides a set of common services for information, security, and management. However, to date, Grid middleware gives little attention to software development issues such as compilation, deployment, and debugging.

Due to the distributed and heterogeneous nature of test beds, developing Grid applications is more complex than for traditional local applications. Grid programming is complicated by the lack of sophisticated tools as described in a quote by Balle and Hood in 2004 [3], "Unfortunately we are not able… to understand why so few people are running or developing programs for the Grid. Our guess is that developing applications for the Grid is too hard and too time consuming. By surveying the landscape for Grid-aware tools, we realize that few robust tools are currently available."

There is a recent surge of interest that addresses the issue of Grid development tools, with initiatives such as g-Eclipse [4] and GDT [5]. These projects utilize extensible IDEs (e.g. Eclipse [6] and NetBeans [7]) to provide powerful and cohesive development environments for the benefit of both Grid programmers and users. IDEs have a steep learning curve and are difficult to master [8][9]. However, various studies have shown that there is an increase in programmer's productivity once a particular environment has been mastered [10][11][12]. IDEs improve productivity in three areas. First, in the area of quality, they provide powerful tools, for example, an automatic syntax checker. Second, through features such as automatic recompilation and source code tracking in the debugging phase, IDEs improve productivity in the area of code management. Third, they provide functions such as code-generation support and refactoring that can enhance programmer's efficiency [10][13].

We believe Grid developers can gain tremendous benefit by using IDEs. On the other hand, current "Grid-enabled" IDEs are not as feature rich as their traditional counterparts. For example, debugging Grid applications is still not supported. In this paper, Grid applications are defined as traditional executables that can be submitted to a job scheduler, as opposed to "Grid services" that must be executed in a Web Service Container. We believe the development of traditional Grid applications will continue to grow in spite of the push towards service-oriented architecture (SOA).

This research paper presents an integrated Grid development environment implemented on top of IBM's Eclipse IDE [6] and our application

development framework, Worqbench [14]. It differs from other Grid IDEs by treating Grid resources as first-class objects in the environment and by providing tight integration between the software development process and the test beds. It supports debugging Grid applications by utilizing our Grid debugging service [14][15]. It also implements a multi-view source editor that simplifies programmers' task in writing code for multiple heterogeneous platforms.

The rest of the paper is organized as follows. Section 2 presents Worqbench and Eclipse as the underlying technologies. The architecture and implementation details are described in sections 3 and 4 respectively. In section 5, we outline related work in the area of Grid development environment. Section 6 presents the conclusion.

2. Worqbench and Eclipse

Grid middleware has defined common protocols and services in the domains of information, security, and management. However, common services that support application development do not currently exist. To this end, we have developed a new system called Worqbench [14] which complements existing middleware by providing services for Grid software development.

Worqbench is an integrated and modular framework for application development on the Grid. It is IDE and middleware independent. Thus, Worqbench allows any IDE with appropriate plug-ins to be used with an existing Grid middleware. It provides services and APIs that can be used by clients to package, launch, and debug Grid applications. It has a layered architecture that consists of several main components and a set of modules as interfaces to various middleware and IDEs. The framework links an IDE to Grid middleware in a three-tier model as depicted in Figure 1.

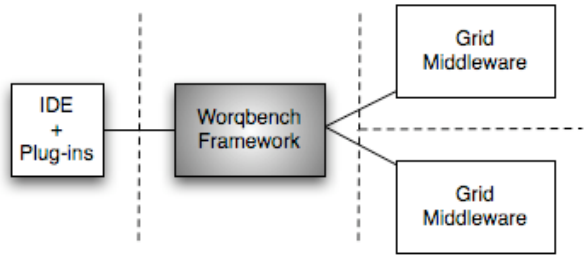


Figure 1. Three-tier model

Worqbench allows the aggregation of Grid resources with different middleware and architectures into a single Grid test bed accessible from the programmer's preferred development environment. The architecture of the framework is shown in Figure 2. The key services are namely packaging, launching, and editing services.

Packaging Service: The Packaging Service is responsible for building or compiling a Grid application on remote resources. Typically, it does this by transferring a compressed file containing source code data to remote machines, setting up and initializing the appropriate directories, and executing a build script to compile the application. The service assumes that the target resources have a suitable environment, including development tools and libraries, to build the application.

Launching Service: The Launching Service manages the execution of an application on Grid resources. It utilizes the underlying Grid middleware job submission tools. The service supports two modes of execution: normal and debug. The debug mode utilizes a middleware level Grid debugging service [15]. Output from the application is forwarded by the launching service to Worqbench clients. A generic command is provided that allows a client to control the execution of a debugged application.

Editing Service: The Editing Service provides the management of a software project at the file system level. It allows a user to retrieve and to query individual project folders and files. Metadata information of the folders and files such as entry name, parent name, and type can also be queried by using the service.

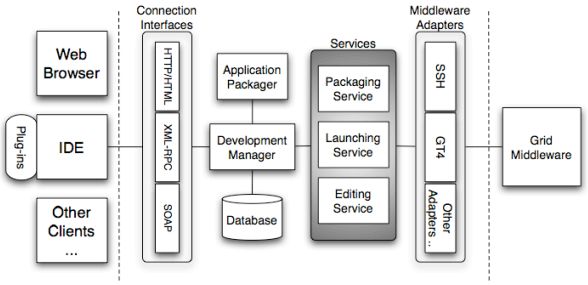


Figure 2. Key components in Worqbench

The description of other components and the internal model of the framework can be found in [14]. Worqbench also provides a web interface portal for administrative tasks such as defining new resources and test beds. However, a web browser is not a suitable environment for application development that requires heavy interaction such as source code editing and debugging which are better provided in an IDE.

We have implemented plug-ins for IBM's Eclipse IDE that utilize Worqbench services. Eclipse is an IDE written in Java. It was originally developed by IBM which gave the code to the Eclipse Foundation under an open-source license. Eclipse can be augmented with new features due to its extensible plug-in system. It

runs on multiple platforms where the Java Virtual Machine is available. Eclipse's GUI performance is fast compared to Java Swing applications because it uses Standard Widget Toolkit (SWT) that utilizes the native GUI libraries of the underlying operating systems. Eclipse supports various programming languages such as PHP, Fortran, Python, and Perl. It also provides an extensive C/C++ support through the CDT (C/C++ Development Tooling) plug-ins. Eclipse has the traditional IDE user-interface with multi-window editors, refactoring, syntax highlighting, and other features commonly found in advanced IDEs.

By implementing Grid development functions as Eclipse plug-ins, traditional application programmers accustomed to Eclipse can develop Grid software without learning another development environment.

3. Architecture of the development environment

Remote Development Tools (RDT) is a set of Eclipse plug-ins that leverage Worqbench services for Grid application development. It consists of four principal components, namely Core, Launch, Editor, and View. The design follows the Eclipse plug-in guidelines [16] by separating the non-UI components (Core and Launch) from the UI components (Editor and View). The architecture of the plug-ins is depicted in Figure 3 while the descriptions are as follows.

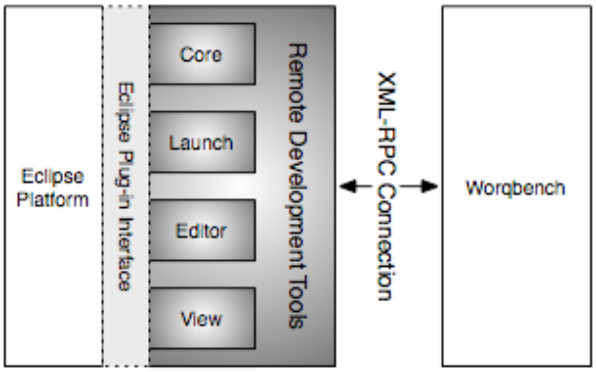


Figure 3. Architecture overview

Core: The Core component is the main link between RDT and Worqbench. All calls to Worqbench services go through the Core component. It provides controllers for Eclipse users to manage resources, resource sets, and Grid application projects.

Launch: The Launch component is responsible for the execution and debugging of a Grid application. It is linked with the built-in Eclipse launching framework and debug model. The component communicates with Worqbench's packaging and launching services.

Editor: The Editor component provides editing service for source files in RDT. The component utilizes low-level Worqbench APIs to implement a multi-view code editor. It can display multiple views of the source code based on keywords given by a user.

View: The View component is responsible for various user-interface in RDT. It implements table views to show resources, resource sets, and projects. It provides console windows and Eclipse preference windows for RDT configuration.

3.1. Internal model

Worqbench maintains an internal model of the system and it defines entities such as *resources*, *resource sets*, *projects*, and *sessions* [14]. RDT follows this system model according to the Model-View-Controller (MVC) design pattern. RDT implements controllers in the *Core* component and the corresponding user-interface in the *View* component for the entities. Unlike Worqbench, RDT does not have a *user* entity since Eclipse is a single-user programming environment.

The *resource* entity represents one distinct machine/resource in the system. A programmer can form a Grid test bed by grouping several resources into a *resource set*. A Grid application project is represented by the *project* entity. The *session* entity represents an RDT development session where the programmer works with a *project* to be tested and executed on a *resource set*.

Traditionally, an IDE deals only with files, folders, and projects as distinct logical objects in the system. It does not distinguish resources since it can only handle one resource, namely the computer where the IDE runs. However, Worqbench and RDT allow a programmer to handle multiple resources from the IDE. By defining *resources* and *resource sets* as part of the system model, the target machines become integral in the IDE. The resources become first-class objects similar to files, folders, and projects. This integration exposes resource heterogeneity in a controlled manner in the software development process. It assists the programmer in writing Grid applications for multiple platforms.

4. Implementation and discussion

RDT is implemented in Java. It uses the XML-RPC interface to communicate with Worqbench. The Core component, which is an XML-RPC client, translates the requests and responses from XML to Java objects required by the plug-in. Worqbench URL address and the user credentials can be specified in the Eclipse preferences window. Section 4.1 describes the implementation of common IDE features for Grid application development, for example debugging and console output. Section 4.2 discusses tagging as a

mean of providing metadata information in RDT. Section 4.3 presents a multi-view code editor that aids Grid programmers in code development for heterogeneous resources.

4.1. Common IDE features

We have developed a Grid debugging service [15] that is compliant with the Web Service Resource Framework (WSRF), which is the standard for Grid middleware architecture [17][18]. The service complements existing middleware by providing a standard set of application programming interfaces (APIs) suitable for debugging and testing computational Grid applications. These debug APIs are utilized for debugging by the Worqbench Launching Service.

The RDT Grid debugging plug-in acts as an intermediate layer between Eclipse and Worqbench debug models. It translates, for example debug events and breakpoints between the two models. The plug-in is currently adapted for C and C++ applications due to Worqbench's limited support for other languages. However, the plug-in extends the generic language-independent Eclipse platform debug model rather than the Eclipse CDT debug model, thus, the difficulty in adapting it to support other languages is mitigated.

The RDT Grid debugging plug-in, which is a part of the RDT Launch component, allows a programmer to debug Grid applications running on remote resources from the Eclipse IDE on the programmer's desktop. It supports common debugging operations such as inspecting variables; adding or removing breakpoints; stepping or resuming the execution; code highlighting of the current line; and so forth. An example of a debugging session is depicted in Figure 4.

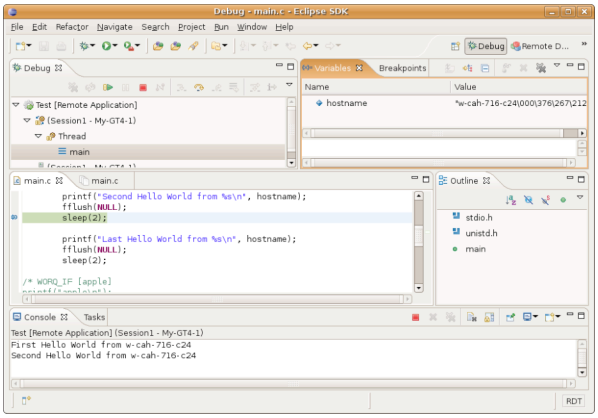


Figure 4. Grid debugging service in Eclipse

To execute a Grid application, the programmer opens the Eclipse launch configuration dialog as depicted in Figure 5. The programmer needs to specify the development session, the Grid application project, and the resource set on which the application will be executed. There is also an option to perform remote build before the execution.

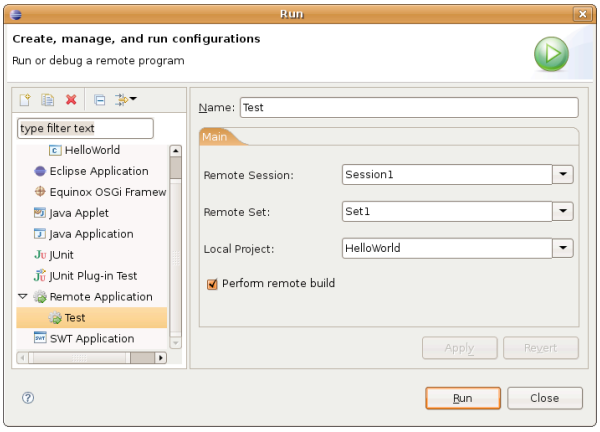


Figure 5. Eclipse launch configuration dialog

The output from executing or debugging a Grid application is displayed in the Eclipse console windows. Each window displays the program output from one resource. The console window also accepts user input for the application.

RDT has its own Eclipse perspective, the Remote Development perspective that displays windows and editors related to Grid application development. RDT implements various Eclipse views that provide management functions for remote resources, resource sets, and sessions. Figure 6 depicts the RDT perspective.

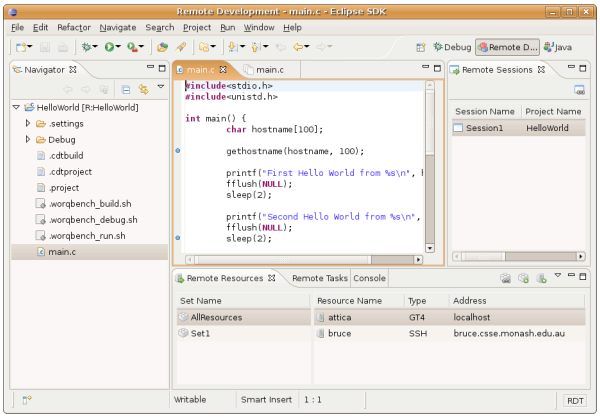


Figure 6. RDT perspective

4.2. Metadata and tagging

Although Grid middleware defines common security, management, and information services and

offers a uniform way for executing Grid applications, it does not solve the problem of architectural heterogeneity. The problem stems from the fact that Grid test beds may consist of a mix of resources with different operating systems and architectures, which must be taken into account by application developers. There are two general approaches. One approach is to employ a software layer to manage the underlying differences. This way requires the applications to be written in high-level languages that have portable virtual machines and interpreters such as Java and Python. However, it may not fully exploit advantages offered by some specific computer architectures and in practice, majority of Grid applications are still written in system languages such as C/C++ or Fortran. The second approach, which is taken by this research paper, is to expose the differences (i.e. the system properties of the resources) but provide tools for managing the heterogeneity. Developers can write applications in low-level languages and use these tools to assist them in tailoring the code to specific architectures.

Worqbench and RDT employ tagging to describe information about entities such as resources, resource sets, and development projects. A tag is a relevant keyword or term associated with or assigned to an entity, thus describing the item and enabling keyword-based classification of information [19][20]. An item can be tagged with multiple keywords which are defined arbitrarily and informally by users. For example, a Sun workstation can be tagged with "`sun sparc solaris`" or a Linux machine can be tagged with "`i686 redhat kernel-2.6.20 smp development-machine`".

Recently, tagging systems have grown in popularity on the web [20]. Many web applications such as Flickr, Delicious, and Google allow users to tag resources (e.g. photos, bookmarks, and emails) with their own keywords. Tagging is user-centric or consumer-centric as opposed to producer-centric. It lets users to associate values or properties as perceived by them to the resources. As far as we are aware of, tagging has never been applied in relation to computer machines and test beds. In the context of Grid computing, this approach has the potential to improve discovery and information systems for Grid resources. Programmers and users will be able to discover "soft facts" (e.g. ranking, reputation, popularity, etc.) in addition to "hard facts" (e.g. operating system versions, architecture types, memory sizes, etc.). We plan to develop interfaces to various information services such as Globus Toolkit Monitoring and Discovering Systems (MDS) to populate tag fields of resources with factual information automatically and without any user actions, for example: "`machinetype:i686-pc ostype:linux-gnu`". Currently, this information must be entered manually in tag fields.

The idea of tagging, however, is not new in the domain of software development. Conditional compilations and "`ifdefs`" which are the traditional ways to mark portions of code with metadata information can be considered as tagging systems. For example in "`#ifdef WIN32`", the keyword "WIN32" can be regarded as a tag, thus, tagging is a natural and familiar concept to application developers.

A system-wide tagging for source code and resources is employed in Worqbench and RDT. It opens up possibilities for various interactions between application source code and target resources. For example: highlighting source files that need to be updated when a new resource is added to the test bed by looking at the resource tags, listing code portions that will not be executed in selected computers by examining the resource and code tags, and so forth.

4.3. Multi-view code editing

It is common for Grid programmers to write software for multiple platforms. However, it is not a trivial task. Different operating systems and computer architectures may introduce different function calls, arguments, and interfaces. Furthermore, some platforms may not have functions provided by others. Writing multi-platform applications introduces variations in the source code that are typically handled in three ways: project level, file level, and code level. At the project level, a programmer creates one distinct application project for each platform. At the file level, the variations are handled by creating different source files, e.g. `utility-win32.c` and `utility-unix.c`. Files common for one platform can then be placed in one directory/folder. The third way, which is the typical method, is to put the variations in the source code itself by employing pre-processor and macro directives, e.g. `#define` and `#ifdef`. A programmer normally employs a mix of the second and third methods.

Macro and pre-processor directives are difficult and error-prone to manage [21]. Adding a new unique platform requires the programmer to find, inspect, and possibly change every occurrence of the directives. In addition, directives also make understanding the source files difficult because of the additional code variations [21]. Various tools and editors have been developed in the software engineering domain to manage such directives [22][23][24].

In contrast to existing approaches, we have developed a multi-view code editor for managing code variations based on the concept of tagging. The editor can show and hide source code depending on user-given keywords or tags. It has one base view and

multiple sub views. The base view shows the full raw source code while the sub view shows only the relevant lines of the code based on the tag. An example to illustrate this is given below.

We define two new directives `/* WORQ_IF … WORQ_END */` and `// WORQ_IF … // WORQ_END` as code markers. We avoid overloading `#define` and `#ifdef` since they can also be used by programmers for non-platform-variation purposes, e.g. for a debugging purpose. `WORQ_IF` accepts a Boolean expression as the condition, for example "`!linux && (sparc || i386)`" marks the code that is relevant when the tag contains `sparc` or `i386` but not `linux`. Consider, for example, the following source file:

```
#include<stdio.h>
int main() {

/* WORQ_IF [linux && sparc]
printf("Linux on a Sparc machine\n");
WORQ_END */

// WORQ_IF [windows]
printf("Windows\n");
// WORQ_END

return 0;
}
```

When the file is opened with the tag "`linux sparc`", the editor opens a new sub view with the following code:

```
#include<stdio.h>
int main() {

printf("Linux on a Sparc machine\n");

return 0;
}
```

If the given tag is "`linux`", the editor shows the main body with only the return statement since the condition of the `WORQ_IF` is "`linux && sparc`". Similarly, when the given tag is "`windows`", the sub view shows:

```
#include<stdio.h>
int main() {

printf("Windows\n");

return 0;
}
```

The editor works with a single instance of the file and it does not keep a separate copy of the file for each sub view. A programmer can edit the source text in a view and the changes will be saved in the file. The problem of concurrency is avoided by allowing only one view to be active and editable at one time. Figure 7 depicts the base view of the editor showing the raw source code. Figure 8 depicts the sub view of the editor with the tag "`linux sparc`".

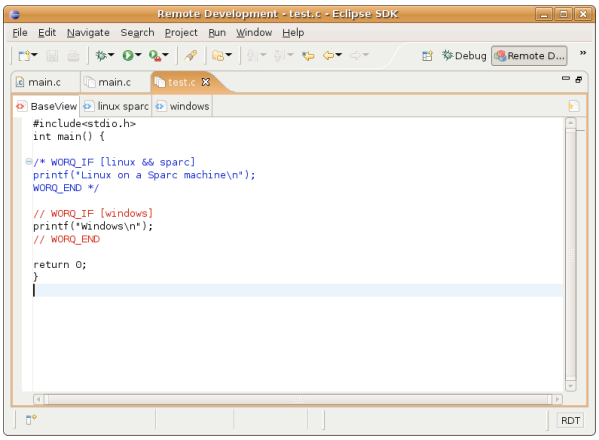


Figure 7. Base view of the editor

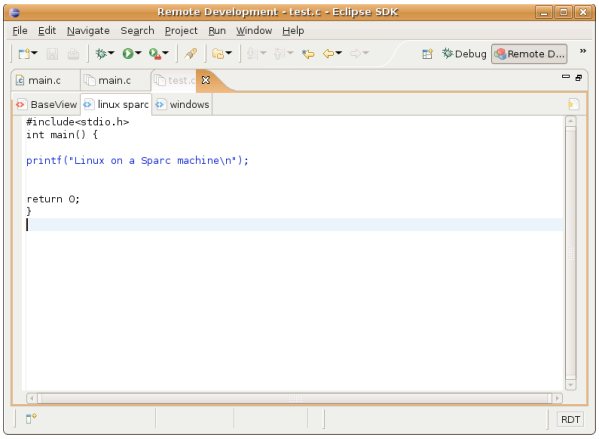


Figure 8. Sub view of the editor

5. Related work

There are several integrated development tools which are specifically designed for writing Grid applications. Some examples of them that represent the current state of research are: GT4IDE [25], GriDE [26], Grid Development Tools [5], and g-Eclipse [4].

GT4IDE [25] is an integrated development environment specifically suited to the needs of Globus Toolkit 4 (GT4) programmers. It is built on top of Eclipse IDE. The main target of GT4IDE is the development of Grid applications as services specifically GT4 services. It presents an environment

that seamlessly integrates all the steps from coding to deployment. It provides the necessary build tools and automatic generation of WSDL (Web Services Description Language) files for GT4 Grid services. Due to the early stage of the implementation, GT4IDE still lacks necessary features such as a Grid debugger and profiler.

GriDE [26] from Asia Pacific Science & Technology Center (APSTC) is a Grid-enabled IDE utilizing NetBeans as the base IDE. NetBeans [7] is an open-source IDE from Sun Microsystems. Similar to Eclipse, it has standard features commonly found in traditional IDEs: refactoring support, syntax highlighting, visual editors, and an extensible plug-in system. GriDE provides a traditional programming environment through NetBeans and a set of Grid-specific tools: a simple Grid workflow editor, a resource browser, a job submission tool, and a job monitoring tool. GriDE's Grid workflow editor can be used to construct simple workflow like batch job and single job executions. Users can browse available Grid resources using the resource browser and with the job submission tool users can submit Globus Toolkit 2 jobs directly from the IDE. Functionalities like Grid debugging and complex Grid workflows are still unimplemented.

Grid Development Tools (GDT) [5] is a set of plug-ins for service-oriented application development in the Eclipse IDE. It is part of the Marburg Ad-hoc Grid Environment (MAGE) which is a WSRF-compliant Grid middleware solution developed in the University of Marburg in Germany. GDT follows a model-driven approach by dividing the development process into three separate model layers and automatically transforming models from one layer into the other. Similar to GT4IDE, GDT is geared towards Grid services and it supports GT4 and MAGE services. GDT utilizes the standard remote Java debugging in Eclipse to support interactive service debugging.

The g-Eclipse [4] project aims to build an integrated workbench tool for Grid computing on top of the Eclipse IDE. It is developed by a collaboration of several European universities under the g-Eclipse consortium. g-Eclipse provides three distinct Eclipse perspectives for Grid users, operators, and developers. The user perspective allows a Grid user to submit and monitor computational jobs. The operator perspective offers tools for managing and maintaining Grid resources, storage elements, and virtual organizations. The developer perspective allows a Grid programmer to code, execute, and debug Grid applications. g-Eclipse is still under active development, the latest prototype release contains Grid management tools for authentication, resources, jobs, and data.

Different to all the tools described, our approach consists of two parts, Worqbench and RDT. Worqbench implements the back-end services and APIs for Grid application development while the user-interface and high-level tools are provided by RDT in the form of Eclipse plug-ins. This modular arrangement allows us to port RDT to other IDEs such as NetBeans without much difficulty. It also enables RDT to work with new Grid middleware without any modification provided that the Worqbench interface to the middleware has been implemented.

6. Conclusion

This research paper presents the design and implementation of RDT, an integrated Grid development environment. It allows e-Scientists and programmers to develop, execute, and test Grid applications on remote resources from local desktops. RDT utilizes Eclipse as the base IDE and leverages Worqbench services for Grid development functions. It provides novel tools to write and debug Grid software.

RDT addresses the challenge of writing software for heterogeneous resources by integrating the test beds with the code development process. In RDT, resources are regarded as first-class objects. It exposes the resource heterogeneity in a controlled manner. RDT provides a tagging system and a multi-view code editor to assist Grid developers in handling the heterogeneity.

RDT, however, is an ongoing research project. Interfaces to various information services such as Globus Toolkit Monitoring and Discovering Systems need to be developed for the tagging system. Even though the implementation of the multi-view code editor is complete, the build and runtime systems still need to be modified to understand the WORQ_IF directive.

7. References

[1] Foster, I., Kesselman, C. and Tuecke, S., The anatomy of the grid: enabling scalable virtual organizations, International Journal of Supercomputer Applications, 2001.

[2] Joseph, J. and Fellenstein, C., Grid Computing, Prentice Hall, 2004.

[3] Balle, S. M. and Hood, R. T., GGF UPDT User Development Tools Survey, March 2004.

[4] g-Eclipse - Access the Grid, http://www.geclipse.org.

[5] Friese, T., Smith, M. and Freisleben, B., GDT: A Toolkit for Grid Service Development, International Conference on Grid Service Engineering and Management, 2006.

[6] Eclipse, http://www.eclipse.org/.

[7] NetBeans, http://www.netbeans.org/.

[8] Reis, C. and Cartwright, R., Taming a Professional IDE for the Classroom, Proceedings of the 35th

SIGCSE Technical Symposium on Computer Science Education, 2004.
[9] Seffah, A., Rilling, J., Investigating the Relationship between Usability and Conceptual Gaps for Human-Centric CASE Tools, p. 226, IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01), 2001.
[10] Case, A. F., Computer-aided software engineering (CASE): technology for improving software development productivity, ACM SIGMIS Database, Volume 17, Issue 1, Fall 1985.
[11] Tsuda, M., Morioka, Y., Takadachi, M. and Takahashi, M., Productivity analysis of software development with an integrated CASE tool. Proceedings of the 14th international conference on Software engineering, 1992.
[12] Glass, R. L., The realities of software technology payoffs. Communications of the ACM. Volume 42 , Issue 2, February 1999.
[13] Frank, R., An Inherent Conflict in Using IDEs in Computer Language Courses. The Proceedings of ISECON 2002, v 19 (San Antonio): 221d. ISSN: 1542-7382, 2002.
[14] Kurniawan, D., Abramson, D., Worqbench: an Integrated Framework for e-Science Application Development, e-Science meeting 2006, Netherlands, December 2006.
[15] Kurniawan, D., Abramson, D., A WSRF-Compliant Debugger for Grid Applications, Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International 26-30 March 2007.
[16] Clayberg, E. and Rubel, D., Eclipse: Building Commercial-Quality Plug-ins, second edition, Addison Wesley, 2006.
[17] Foster I., Kesselman C., Nick J.M. and Tuecke S., The physiology of the Grid: an open grid services architecture for distributed systems integration, Globus Project, 2002.
[18] Czajkowski, K., Foster, I., Frey, J., et al. The WS-Resource Framework, Version 1.0. http://www.globus.org/wsrf/.
[19] Tag (metadata) - Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Tags.
[20] Marlow, C., Naaman, M., et al. HT06, tagging paper, taxonomy, Flickr, academic article, to read, Proceedings of the seventeenth conference on Hypertext and hypermedia, 2006.
[21] Ernst, M. D., Badros, G. J., and Notkin, D. An empirical analysis of C preprocessor use, IEEE Transactions on Software Engineering, 28(12):1146–1170, Dec. 2002.
[22] Kruskal, V., A Blast from the Past: Using P-EDIT for Multidimensional Editing, Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, ICSE 2000.
[23] Livada, P.E., Small, D.T., Understanding Code Containing Preprocessor Constructs, Proc. IEEE Third Workshop Program Comprehension, pp. 89–97, Nov. 1994.
[24] Kuhlbach B., Riediger, V., Folding: An Approach to Enable Program Understanding of Preprocessed Languages. In E. Burd, P. Aiken, and R. Koschke, editors, Proc. of Eighth Working Conference on Reverse Engineering; WCRE'01, pp. 3–12, IEEE Computer Society, 2001.
[25] Globus Service Build Tools - GT4IDE, http://gsbt.sourceforge.net/.
[26] GriDE, https://gride.dev.java.net/.

An IDE Framework for Grid Application Development

Donny Kurniawan and David Abramson
Monash e-Science and Grid Engineering Lab
Faculty of Information Technology, Monash University
{donny.kurniawan, david.abramson}@infotech.monash.edu.au

Abstract

Grid computing enables the aggregation of a large number of computational resources for solving complex scientific and engineering problems. However, writing, deploying, and testing grid applications over highly heterogeneous and distributed infrastructure are complex and error prone. A number of grid integrated development environments (IDEs) have been proposed and implemented to simplify grid application development. This paper presents an extension to our previous work on a grid IDE in the form of a software framework with a well-defined API and an event mechanism. It provides novel tools to automate routine grid programming tasks and allow programmable actions to be invoked based on certain events. Its system model regards resources as first-class objects in the IDE and allows tight integration between the execution platforms and the code development process. We discuss how the framework improves the process of grid application development.

1. Introduction

A grid infrastructure can be defined as a system that coordinates resources that are not subject to centralized control using standard, open, general-purpose protocols and interfaces to deliver non-trivial qualities of service [1]. Grids that comprise several organizations are inherently large-scale, distributed, and heterogeneous. These attributes and other concerns such as data access and authentication must be confronted by grid application developers. To address some of these concerns, the grid community has developed specific middleware that provides a uniform set of services for security, information, and management. Middleware can be defined as the software layer between the operating system and applications, providing common high-level functions required by applications to operate correctly. In a grid, the middleware is used to hide the heterogeneous nature and provide users and applications with a homogenous and seamless environment by supplying a set of standardized interfaces to a variety of services [2]. However, to date, grid middleware gives little attention to software development issues such as compilation, deployment, and debugging.

Grid application development is concerned with the process of writing, testing, and debugging grid-based software. It is different from the traditional counterpart due to the scale and heterogeneity of the target platforms. Rather than targeting one type of computer architecture and operating system, grid software development must take into account the heterogeneous and distributed nature of grid infrastructure. Traditional programming tools are not sufficient to handle the complexity and scale of the infrastructure. Grid application development necessitates programming tools and services that are designed and engineered for grid computing. In this paper, grid applications are defined as traditional executables that can be submitted to a job scheduler, as opposed to "grid services" that must be executed in a web service container. We believe the development of traditional grid applications will continue to grow in spite of the push towards service-oriented architecture.

Recently, there is a surge of interest in grid integrated development environments (IDEs) with initiatives such as g-Eclipse [3] and GDT [4], to name a few. These projects augment IDEs such as Eclipse [5] and NetBeans [6] with plug-ins that enable the tools to handle grid application development. An integrated development environment (IDE) combines various programming tools such as a source code editor, a compiler, and a debugger into one cohesive environment [17]. The integration allows the tools to work together seamlessly, for example, the debugger can highlight source code errors in the editor and the compiler can be invoked automatically when the code changes. We believe grid developers can gain significant benefit by using IDEs.

We have designed and implemented an infrastructure for supporting e-Science and grid application development. The infrastructure consists of a grid debugging service (GridDebug) [7], a multi-tiered programming middleware (Worqbench) [8], and an integrated grid development environment (RDT) [9]. RDT, which stands for Remote Development Tools, is a set of Eclipse IDE plug-ins that allows the IDE to be used for grid application development. This research paper presents and discusses our work on an *extension* to RDT. *We have augmented RDT with an IDE framework with scripting functionality.* It enables

developers to perform and automate various programming tasks, which may become cumbersome and complicated for a large number of heterogeneous grid resources. To achieve this automation, there are certain requirements that need to be met:

- The ability of the system to comprehend various entities such as resources and source file and infer the relationship between these objects.
- A runtime programmable facility that allows the developers to call or change the functionality of the IDE during its execution.

RDT has been designed and implemented to meet these requirements, specifically, the features are:

- A system model with an abstraction of grid resources and sets as first-class objects similar to files, folders, and projects.
- A system-wide tagging system for attaching metadata information to various entities such as resources and projects.
- An application programming interface (API) that enables RDT and the IDE to be accessed programmatically during runtime.
- An event notification system with user-defined callback methods.

This paper is organized as follows: Section 2 presents related work in the area of grid integrated development environment. Section 3 describes the architecture and design of the framework. The implementation details are discussed in Section 4. The discussion on how our framework improves grid software development is given in Section 5. Finally, Section 6 gives the conclusion of this research paper.

2. Related Work

There are several integrated development environments, which are specifically designed for writing grid applications. Indicative examples that represent the current state of research are: P-GRADE [10], g-Eclipse [3], GDT [4], and Introduce [11].

P-GRADE (Parallel Grid Runtime and Application Development Environment) [10] is a high-level graphical environment to develop parallel applications for parallel systems and grid. It consists of an editor (Gred) for a graphical language (Grapnel), a debugger (Diwide), a profiler (Prove), and application monitoring tools (GRM and Mercury). In P-GRADE, rather than directly write an application in C/C++ or Fortran, developers write the program graphically in Grapnel using Gred. Grapnel is a hybrid programming language that uses both textual and graphical representations to describe the application. P-GRADE converts the Grapnel code to C/C++ or Fortran code, links it with the necessary library, and compiles it into an executable.

g-Eclipse [3] is an integrated workbench tool for grid computing built on top of the Eclipse IDE. It has been developed by the g-Eclipse consortium which is a collaboration of several European universities and companies. g-Eclipse distinguishes three distinct user roles, namely application users, grid operators, and developers. It provides different Eclipse perspectives for each role. The user perspective allows a grid user to submit and monitor computational jobs. The operator perspective offers tools for managing and maintaining various entities such as grid resources, storage elements, and virtual organizations. The developer perspective allows a grid programmer to code, execute, and debug grid applications.

Grid Development Tools (GDT) [4] is a set of Eclipse plug-ins for service-oriented grid application development. GDT is targeted at non-grid programmers and it allows them to rapidly develop grid applications without knowing the intricacies of the middleware. GDT follows a model-driven approach by dividing the development process into three separate model layers and automatically transforming models from one layer into the other. GDT is geared towards service-oriented grids and it supports the development of Globus Toolkit 4 and MAGE services.

Introduce [11] is an open-source extensible toolkit that provides tools and an environment for the development and deployment of strongly typed, secure grid services. These services consume and produce data with explicit and well-defined types. It enables syntactic interoperability among grid services. Introduce leverages Globus Toolkit 4 as the underlying grid middleware to facilitate the development of WSRF-compliant services.

These existing tools benefit grid programmers by providing development environments that ease and simplify the process of grid application development. They take advantage of extensible non-grid IDEs to build software tools for grid computing. They implement specific grid functions such as job submission tools and resource browsers that let programmers inspect available grid resources directly from the IDE. These grid development tools provide extension points and mechanism that allow them to be extended through IDE plug-ins or enhancements. The plug-ins need to be written, compiled, and packaged according to the formal procedure as defined by the IDE.

Different to all the IDE tools described, we have designed and implemented a user-level IDE framework for grid application development. RDT has been extended with a framework with a well-defined API and an event mechanism. The framework enables RDT to be accessed programmatically and allows scriptable actions to be invoked based on certain events. These actions provide automation functions for routine grid

programming tasks. The RDT system model provides an abstraction of grid resources and sets as first-class objects in the IDE. RDT presents an alternative mechanism for improving and altering the behavior and functionality of the IDE by utilizing small and well-written scripts. There are certain advantages of using a high-level scripting language such as late binding, reflection support, expressive syntax, and dynamic typing. This scripting mechanism is different from the formal approach of extending the IDE by writing and packaging elaborate plug-ins.

3. Architecture and Design

RDT is designed on top of Worqbench [8], an integrated and modular framework for grid application development. Worqbench provides services and APIs that can be used by clients to package, launch, and debug grid applications. It allows the aggregation of grid resources with different middleware and architectures into a single grid resource set accessible from the programmer's preferred development environment. Worqbench links an IDE to grid middleware in a three-tier model as depicted in Figure 1. A detailed description of Worqbench and its internal model can be found in [8][9].

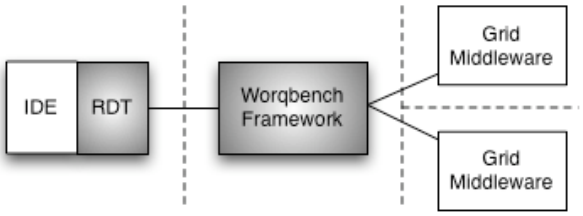


Figure 1. Worqbench three-tier model

RDT is not a replacement for an integrated development environment. Rather, it is designed as an extension to an IDE. Thus, it depends on an IDE for functions such as source code editing, project management, debugging, and so forth. It has a high-level architecture and the design is IDE-independent. RDT consists of four main components in two software layers: user interface (UI) and non user interface layers. The layer division ensures modularity and simplifies the integration of RDT into an IDE [13]. The UI tools can be customized according to the base IDE whilst retaining the underlying RDT models. The UI layer consists of two components, View and Editor, that aim to cover principal development tasks such as source code editing and resource browsing. The non-UI layer is composed of a framework model component (Core) and an application execution manager (Launch). A non-grid IDE is designed in general to handle program execution on one platform, namely, the local machine where the IDE runs. The RDT Launch component extends the execution mechanism to handle a number of distinct remote platforms simultaneously. The architecture of RDT is depicted in Figure 2 while descriptions of the components are given as follows:

Core. The Core component specifies a framework and API with models, methods, and data structures using an object-oriented paradigm. Classes are defined to represent entities such as resources, resource sets, projects, sessions, and tasks. It is the main component of RDT which also handles the event mechanism and various event callbacks.

Launch. The Launch component is responsible for managing the execution and debugging of applications. It depends on the underlying IDE launching framework and debug model.

View. The View component is responsible for various user-interface in RDT. It provides the implementation of views such as data tables to show resources, resource sets, and projects. The component also implements preferences user-interface for RDT configuration.

Editor. The Editor component provides the implementation of a multi-view code editor. It can display multiple views of the source code based on user-given keywords or tags. The editor simplifies the process of writing grid applications for multiple architectures and operating systems. More information about the editor can be found in [9].

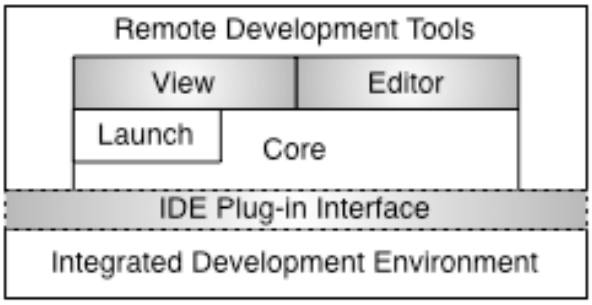


Figure 2. RDT architecture overview

Section 3.1 gives the description of the RDT framework, system model, and API. Sections 3.2 and 3.3 describe the RDT tagging system and event mechanism.

3.1 System Model and API

Traditional IDEs are geared towards local application development and normally target the platform in which they are installed on. Thus, IDEs are mainly concerned with software projects, folders, and files as opposed to target resources. RDT extends this conventional model to include grid resources and resource sets as first-class objects in the IDE. It enables

the IDE framework to recognize the relationship that exists between software projects and various grid resources; and invoke user scripts based on this association. Figure 3 depicts an abstract model and the relationship of various RDT objects.

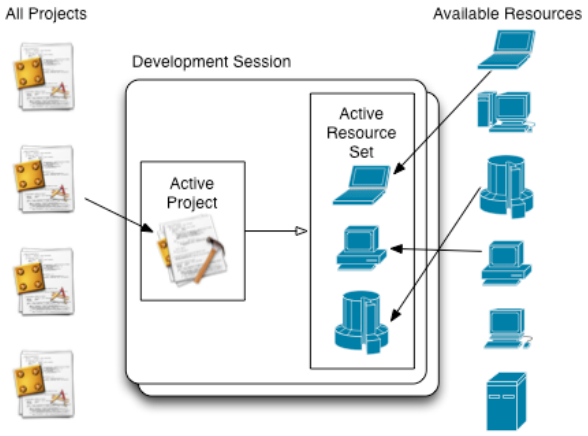


Figure 3. Abstract model of RDT

We define a development session that explicitly links an active grid resource set with an active project. An active resource set is a collection of resources where a programmer intends to develop for. The resources are selected from a pool of available machines. A project that the programmer is currently working on is called the active project. RDT operates with development sessions. With the information of the intended grid execution platforms in a session, RDT can perform various automatic or scripted programming tasks on behalf of the programmer.

A tight integration between the process of writing code and the execution platforms facilitates several development scenarios. For example, programmers can conduct performance testing on multiple platforms in a test-driven software development process and rapid prototyping of software for heterogeneous grid resources in an agile development method [16].

The framework maintains a system model with classes to represent entities such as resources, resource sets, projects, sessions, and tasks. These entities are described as follows:

Resource. It represents a distinct machine/resource in the system on where a software development project can be built and the resultant application can be executed. It is assumed that the remote machine has the necessary environment, including compilers and libraries, to build the application.

Resource Set. A number of resources can be grouped together to form a resource set or a transient grid testbed which becomes a deployment or execution target of a project. The framework defines a special set called "All Resources" that contains all resources in the system.

Project. It represents an RDT project which is in essence an IDE-specific project with additional RDT metadata information such as user-specified tags or keywords.

Session. It represents a development session where a user writes an application in one project to be tested and executed on one resource set. RDT operations such as remote compilation, execution, and debugging require a session object.

Task. It represents a remote operation invoked on a session object. It stores the status, type, and output of the operation. For example, a user invokes a remote compilation on a session object. This operation is represented by a task object which can be inspected by the user for its attributes (e.g. status and output).

Various managers, namely ResourceManager, SetManager, ProjectManager, SessionManager, and TaskManager, handle the creation, deletion, and administration of these entities.

RDT defines a set of API methods that allows programmers to write scripts that utilize these entities and perform various IDE functions. Scripts can be coded to automate and repeat many tedious grid programming tasks, such as software compilation and unit testing, across a variety of resources. Thus, this automation enables the programmers to conduct grid application development for a large number of target platforms efficiently, a process which is significantly more complex than when software is developed for a single platform. Moreover, because we use scripting, the user is able to tailor these functions to cater for variations in their particular grid infrastructure and they are not locked into a standard implementation. Importantly, the RDT framework and the API definitions are IDE-independent, however, the implementation and its language binding must follow the specification of the base IDE. A list of the currently implemented interfaces for the Eclipse IDE is given in Section 4. The implementation utilizes the Groovy programming language [12] as its language binding.

3.2 Metadata and Tagging

The system model allows the framework to recognize the relationship between various entities. However, it does not specify the semantics of the relationship. As a solution, RDT employs a tagging system that allows metadata information to be attached to RDT entities. A tag is a relevant keyword or term associated with or assigned to an entity, thus describing the item and enabling keyword-based classification of information [14]. An entity can be tagged with multiple keywords which are defined arbitrarily and informally by users. For example, a Linux machine can be tagged

with "`i686 kernel-2.6.20 smp development-machine condorpool`" or a Sun workstation can be tagged with "`sun sparc solaris globus`". Source code can also be tagged, for example, a source file for accessing a WS-GRAM service can be tagged with "`gt4 webservice`". This system-wide tagging system facilitates the sharing of metadata information between source code files and grid resources; and enables the framework to understand the relationship semantics.

3.3. RDT Event Mechanism

IDEs employ an event mechanism for certain functions. For example, the editor can be instructed to highlight a particular line of code when a breakpoint-hit event occurs or the compiler can be set to build a project when a source file is saved. Similarly, RDT provides an event mechanism based on the framework system model with user-definable callback methods. A user can write the methods to perform various checks and operations when specific events occur. For example, a warning can be displayed when a grid resource tagged with "`windows`" is added to a "`unix`" resource set, source files that need to be updated are highlighted when a new grid resource is added to a "`test bed`" set, and so forth. The event mechanism has been designed to be generic and IDE neutral. Table 1 lists the currently supported events. Several examples of the event callback methods are given and discussed further in Section 4.

ProjectChangedEvent	SetDestroyedEvent
ProjectCreatedEvent	FileChangedEvent
ProjectDestroyedEvent	FileCreatedEvent
ResourceAddedEvent	FileDestroyedEvent
ResourceRemovedEvent	FolderChangedEvent
ResourceCreatedEvent	FolderCreatedEvent
ResourceDestroyedEvent	FolderDestroyedEvent
SetCreatedEvent	

Table 1. RDT Events

4. Implementation Details

We have developed Eclipse RDT, a set of plug-ins that provide the implementation of RDT for Eclipse [5] which is an open-source IDE written in Java. Eclipse, like a number of other IDEs such as NetBeans and Visual Studio, can be augmented with new features due to its extensible plug-in system. It runs on multiple platforms where the Java Virtual Machine is available. It supports various programming languages such as Fortran, Python, Perl, and C/C++. It uses the Standard Widget Toolkit (SWT) that utilizes the native GUI libraries of the underlying operating systems. Eclipse has the traditional IDE user-interface with multi-window editors, refactoring, syntax highlighting, and other features commonly found in advanced IDEs. Eclipse RDT is implemented according to the Eclipse plug-in guidelines [15]. It is divided into two software layers, UI and non-UI; and it consists of several Java packages.

Eclipse RDT features an embedded interpreter for the Groovy programming language [12]. Developers utilize the framework API and event mechanism by writing scripts and event callback methods in Groovy. It is a dynamic object-oriented scripting language for the Java platform. Groovy has features similar to Python, Perl, and Ruby such as dynamic typing; late binding; reflection; native syntax for lists, maps, and regular expressions; and closures. It has a simpler syntax than Java, support for various markup languages, and powerful processing primitives. In addition, it can invoke and seamlessly integrate with all existing Java objects and libraries.

The graphical user interface aspect and the multi-view code editor of Eclipse RDT have been the subject of another research paper [9] and they are not repeated here. The following two sections present the implementation of Eclipse RDT framework. Section 4.1 describes the Eclipse RDT API whilst Section 4.2 discusses the implementation of Eclipse RDT event mechanism.

4.1 Eclipse RDT API

The Eclipse RDT application programming interface is implemented as Java classes that can be called and accessed directly from within Groovy scripts. There are two ways to access the framework API:

- Through a Groovy shell view. It is a Groovy command-line interpreter that is contained in a special Eclipse window view. It allows a programmer to evaluate Groovy expressions, invoke methods, define classes, and run Eclipse RDT scripts that alter the behaviour and functionality of the IDE. The shell view is similar to the Python interpreter program or the Interactive Ruby Shell (IRB).
- Through an Eclipse event preferences window. It is an event configuration dialog where a developer can write and save RDT scripts for event callback methods. The scripts are standard Groovy scripts that can call and access methods in the framework API. Eclipse RDT will run the event scripts based on the occurrence of corresponding events.

Table 2 lists the currently implemented interfaces for the RDT entities and managers. In addition to RDT API, built-in Eclipse APIs including the SWT GUI widgets are also callable within the Groovy scripts.

```
IResource:
String getTags()
void setTags(String tags)
String getResourceName()
Integer getResourceId()
String getResourceType()
String getResourceAddr()
Integer getResourcePort()
void setResourceName(String newName)
void setResource(Integer newResource)

ISet:
String getTags()
void setTags(String tags)
String getSetName()
Integer getSetId()
IResource[] getResources()
void addResource(Integer resId)
void removeResource(Integer resId)
void refresh()
void setSetName(String newName)

IProject:
String getTags()
void setTags(String tags)
String getProjectName()
Integer getProjectId()
void setProjectName(String newName)

ISession:
String getTags()
void setTags(String tags)
String getSessionName()
Integer getSessionId()
IProject getProject()
ISet getSet()
void setSessionName(String newName)
void setProject(Integer newProject)
void setSet(Integer newSet)

ITask:
String getTaskName()
Integer getTaskId()
String getTaskType()
String getTaskStatus()
String getTaskOutput()
void refresh()
ISession getSession()
IResource getResource()

ResourceManager:
void refresh()
IResource newResource(String resName, int
res_id)
void destroyResource(Integer id)
IResource[] getResources()
IResource getResource(Integer id)
IResource getResource(String resName)

SetManager:
void refresh()
ISet newSet(String setName)
void destroySet(Integer id)
ISet[] getSets()
ISet getSet(Integer id)
ISet getSet(String setName)

ProjectManager:
void refresh()
IProject newProject (String projectName)
void destroyProject(Integer id)
IProject[] getProjects()
IProject getProject(Integer id)
IProject getProject(String projName)

SessionManager:
void refresh()
ISession newSession (String sessionName,
int proj_id, int set_id)
void destroySession(Integer id)
ISession[] getSessions()
ISession getSession(Integer id)
ISession getSession(String sessName)

TaskManager:
void refresh()
void destroyTask(Integer id)
void destroyAllTasks()
ITask[] getTasks()
ITask getTask(Integer id)
ITask getTask(String taskName)
```

Table 2. Eclipse RDT API

The creation, deletion, and administration of RDT entities are handled by a number of RDT managers, which are implemented as singleton objects. The managers construct objects such as `IResource`, `ISet`, `IProject`, `ISession`, and `ITask` objects, which can be used directly by calling their respective methods in an object-oriented manner. RDT entities share a number of matching attributes, specifically, `name`, `id`, and `tags` (with the exception of the `tags` attribute for the `ITask` entities). These attributes indicate the name, unique global id, and tags of the entities. It is possible to extend the API by implementing more entity attributes or by providing composite methods that perform certain higher-level functions. Eclipse RDT depends on Worqbench services for certain functions such as grid resource authentication and user authorization. The following code listing shows an example of utilizing Eclipse RDT API.

```
allResources =
   ResourceManager.getResources()
allMonashResources = []

for (resource in allResources) {
   if (resource.getResourceAddr() ==~
       /monash.edu.au/)
       allMonashResources.add(resource.ge
       tResourceName())
}
```

```
1  def onFileChanged() {
2     println("File: " + name + " has been changed")
3
4     if (name.endsWith(".sql"))
5        println("Reload the database!")
6  }
7
8  def onResourceAdded() {
9     println("Set is: " + setName + " and Resource is: " + resName)
10
11    vSet = SetManager.getSet(setName)
12    vResource = ResourceManager.getResource(resName)
13    setTags = vSet.getTags()
14    resTags = vResource.getTags()
15
16    if (resTags.indexOf("windows") != -1 && setTags.indexOf("GT4") != -1)
17       println("Warning: adding a Windows PC to a Globus Toolkit Testbed")
18 }
19
20 def onSetDestroyed() {
21    println("Set: " + name + " has been destroyed")
22
23    allProjects = ProjectManager.getProjects()
24
25    for (project in allProjects)
26       if(project.getTags().indexOf(name) != -1)
27          cmdChangeProjectText(project.getProjectName(), "Inspect Me!")
28 }
```

Figure 4. Examples of event callback methods

```
allMonashResources.eachWithIndex()  {obj,
i -> println "${i}: ${obj}"}
```

This example script prints the name of all grid resources located at Monash University. The script also demonstrates dynamic data typing, regular expressions, and closures in Groovy.

4.2 Eclipse RDT Event Mechanism

Eclipse RDT implements an event mechanism with user-defined callback methods. Internally, it has an event queue, an event listener, and a collection of objects that correlate with RDT events. Eclipse RDT adds an event object to the queue at the completion of related API methods. The event listener checks and removes the event from the queue and invokes the corresponding user-defined callback method. A user writes the callback methods in Groovy in the Eclipse RDT event preferences window. The callback methods have designated names such as `onFileChanged()`, `onProjectDestroyed()`, `onSetCreated()`, and so forth. In addition, there exist special variables that are relevant and valid within the scope of the methods. For example, in the `onSetCreated()` method, a special variable called `name` contains the name of the newly created resource set. A number of examples of event callback methods are given in Figure 4 with the following description:

- The method `onFileChanged()` (lines 1-6) is called when a file has been changed by the user. A special variable called `name` that contains the filename is available within the scope of this method. The data type of `name` is Java String. The example prints a warning to the console output when a file with the extension `sql` is changed.
- The method `onResourceAdded()` (lines 8-18) is called when a resource is added to a set. Two special variables, `resName` and `setName`, are available within this method. On line 11, we initialize a new variable vSet that holds the actual Set object and on line 13, we store `vSet`'s tags in the variable `setTags`. We do the corresponding operations with the resource on lines 12 and 14. On line 16, we perform tests on the tags and print a warning if the conditional is true.
- The method `onSetDestroyed()` (lines 20-28) is called when a set is destroyed with a special variable `name` identifies the set. The example checks all projects for the set's name in the project's tags. If the conditional is true, matched projects are marked with text which is visible in Eclipse's Project Browser.

In the current implementation, built-in Eclipse APIs, data structures, and SWT GUI widgets are callable within the Groovy scripts. Thus, we can display a pop-up dialog box, close Eclipse perspectives, invoke the compile action, run an executable, and so forth based on the occurrence of RDT events.

5. Improving Grid Software Development

Different to other grid IDEs mentioned in Section 2, RDT implements a user-level IDE framework for grid application development. The framework allows programmers to write scripts that utilize RDT and IDE APIs to call or change the functionality of the IDE during its execution. RDT improves the process of grid software development and addresses a number of issues posed by the process such as scalability, efficiency, quality control, and extensibility.

There are many aspects of scalability such as performance, fault tolerance, load management, and code maintainability. However, there is one aspect of scalability that is often overlooked, namely, the ability of the development process to scale as the number of grid resources increases. The RDT framework and its grid system model enable developers to automate many tedious grid programming tasks for a large number of resources. Another aspect related to scalability is efficiency. RDT streamlines grid application development by providing a mechanism that allows programmable actions to be invoked based on certain conditions or grid resources. In addition, the RDT framework and its event mechanism enable check and verification steps to be performed on source code in relation to grid resources and their attributes or tags. The use of a high-level scripting language to drive the development tools is also advantageous since the user can customize the tool functions to cater for variations in their particular grid infrastructure

We believe the framework improves and advances the state of grid programming tools. It simplifies and eases the process of developing grid software for a large number of heterogeneous and distributed resources.

6. Conclusion

Grid application development requires sophisticated programming tools that can handle the complexity inherent in grid infrastructure. Programming tools such as grid development environments have been proposed and implemented. These tools can be further improved by incorporating scripting functionality that allows programmers to perform complex development tasks for a large number of heterogeneous grid resources. This research paper presents an extension to our earlier work, RDT, in the form of an IDE framework. The improvement permits the grid IDE to be scriptable to facilitate the automation of routine and tedious grid programming tasks. The grid system model integrates the execution platforms with the software development process. The paper also presents Eclipse RDT, which is an implementation of the framework for the Eclipse IDE.

RDT can be further improved by defining more event types and API methods. It is possible to improve and simplify the current Eclipse RDT implementation by providing buttons for toolbars and menus that allow users to call and invoke pre-packaged or frequently used RDT scripts directly.

7. References

[1] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. The International Journal of Supercomputer Applications, 15(3), August 2001.

[2] David De Roure, Mark A. Baker, Nicholas R. Jennings, and Nigel R. Shadbolt. The Evolution of the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, Grid Computing: Making the Global Infrastructure a Reality, Chapter 3. Wiley and Sons, 2003.

[3] g-Eclipse - Access the Grid, http://www.geclipse.org.

[4] Thomas Friese, Matthew Smith, and Bernd Freisleben. GDT: A Toolkit for Grid Service Development, International Conference on Grid Service Engineering and Management, 2006.

[5] Eclipse, http://www.eclipse.org/.

[6] NetBeans, http://www.netbeans.org/.

[7] Donny Kurniawan and David Abramson. A WSRF-Compliant Debugger for Grid Applications. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. March 2007.

[8] Donny Kurniawan and David Abramson. Worqbench: An Integrated Framework for e-Science Application Development. In e-Science 2006: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing, December 2006.

[9] Donny Kurniawan and David Abramson. An Integrated Grid Development Environment in Eclipse. In e-Science 2007: Proceedings of the Third IEEE Conference on e-Science and Grid Computing, December 2007.

[10] Peter Kacsuk, Gabor Dozsa, Jozsef Kovacs, Robert Lovas, Norbert Podhorszki, Zoltan Balaton, and Gabor Gombas. P-GRADE: A Grid Programming Environment, Journal of Grid Computing, Volume 1, Issue 2, 2003, Pages 171 - 197.

[11] Shannon Hastings, Scott Oster, Stephen Langella, David Ervin, Tahsin Kurc, and Joel Saltz. Introduce: An Open Source Toolkit for Rapid Development of Strongly Typed Grid Services, Journal of Grid Computing, Volume 5, Issue 4, 2007.

[12] Groovy. http://groovy.codehaus.org/.

[13] Anthony I. Wasserman. Tool integration in software engineering environments. Lecture Notes in Computer Science - Software Engineering Environments, Volume 467, Pages 137-149, 1990.

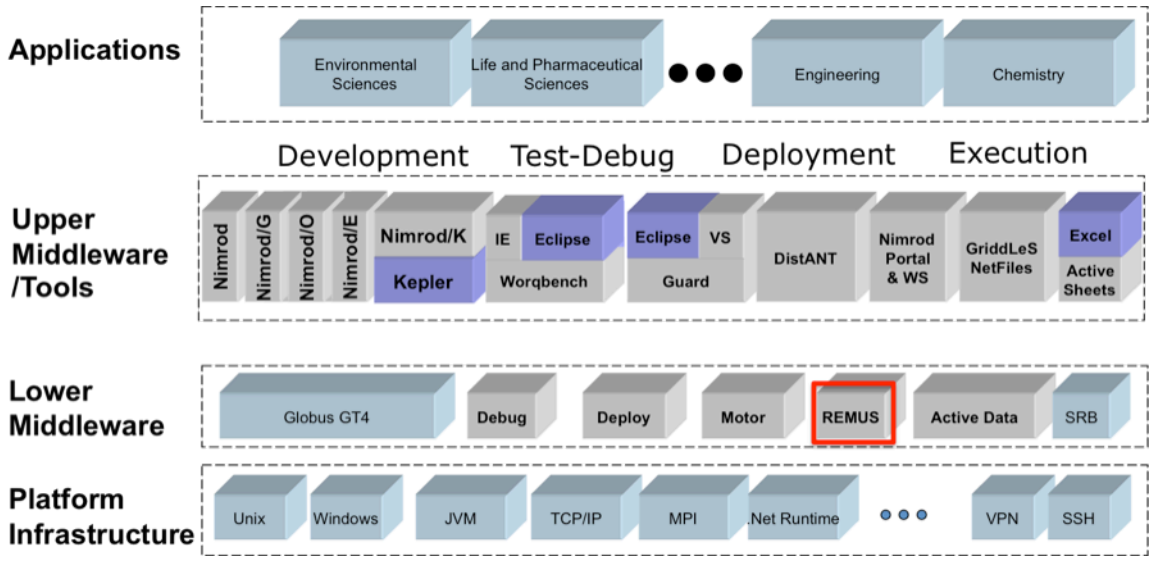
[14] Cameron Marlow, Mor Naaman, Danah Boyd, and Marc Davis. HT06, tagging paper, taxonomy, Flickr, academic article, to read. In Proceedings of the Seventeenth Conference on Hypertext and Hypermedia, 2006.

[15] Eric Clayberg and Dan Rubel. Eclipse: Building Commercial-Quality Plug-ins, second edition, Addison Wesley, 2006.

[16] Craig Larman and Victor Basili. Iterative and Incremental Development: A Brief History. IEEE Computer, Volume 36, Issue 6, 2003.

[17] F. A. Case. Computer-aided software engineering (CASE): technology for improving software development productivity. ACM SIGMIS Database, Volume 17, Issue 1, Fall 1985.

Chapter 13: REMUS and Grid Firewalls



Much of the Grid literature (and software implementation) assumes that Grid resources are connected without security firewalls. However, in the real world, this is not true, and it becomes difficult to connected resources, for example, behind corporate firewalls. In this chapter we discuss techniques for solving this problem by tunneling data through secure channels.

(1) Tan, J., Abramson, D. & Enticott, C. "REMUS: A Rerouting and Multiplexing System for Grid Connectivity Across Firewalls", Journal of Grid Computing, Online version doi:10.1007/s10723-008-9104-1, June 2008 13-1 .. 13-26

Contributions

At the time this work was performed, Tan and Enticott were research associates under my supervision.

J Grid Computing
DOI 10.1007/s10723-008-9104-1

REMUS: A Rerouting and Multiplexing System for Grid Connectivity Across Firewalls

Jefferson Tan · David Abramson · Colin Enticott

Received: 29 June 2007 / Accepted: 21 May 2008
© Springer Science + Business Media B.V. 2008

Abstract The Grid provides unique opportunities for high-performance computing through distributed applications that execute over multiple remote resources. Participating institutions can form a virtual organization to maximize the utilization of collective resources as well as to facilitate collaborative projects. However, there are two design aspects in distributed environments like the Grid that can easily clash: security and resource sharing. It may be that resources are secure but are not entirely conducive to resource sharing, or networks are wide open for resource sharing but sacrifice security as a result. We developed REMUS, a rerouting and multiplexing system that provides a compromise through connection rerouting and wrappers. REMUS reroutes connections using proxies, ports and protocols that are already authorized across firewalls, avoiding the need to make new openings through the firewalls. We also encapsulate applications within wrappers, transparently rerouting the connections among Grid applications without modifying their programs. In this paper, we describe REMUS and the tests we conducted across firewalls using two Grid middleware case studies: Globus Toolkit 2.4 and Nimrod/G 3.0.

Keywords Grids · Grid connectivity · Firewalls · Networks · Network security · Wrappers

Abbreviations

FTP	File Transfer Protocol
DMZ	Demilitarized Zone
GRAM	Grid Resource Access and Management
GSI	Grid Security Infrastructure
GSSAPI	Generic Security Services Application Programming Interface
IP	Internet Protocol
REMUS	Rerouting and Multiplexing System
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VO	Virtual Organization
WS	Web Service(s)

1 Introduction

The Grid refers to "a distributed computing infrastructure for advanced science and engineering" [8]. Grid applications require advanced computations through distributed processing, with an

J. Tan (✉) · D. Abramson · C. Enticott
Faculty of Information Technology,
Monash University, 900 Dandenong Road,
Caulfield East, VIC 3145, Australia
e-mail: jefferson.tan@infotech.monash.edu.au

D. Abramson
e-mail: david.abramson@infotech.monash.edu.au

C. Enticott
e-mail: colin.enticott@infotech.monash.edu.au

explicit focus on supporting *virtual organizations* (VOs). By participating in a VO, institutions are able to share and coordinate computations, commonly with the Internet as the underlying communication infrastructure. Unfortunately, on the Internet, there is always the risk of unauthorized access to data, transactions or computational resources. For this reason, networks on the Internet are often protected with firewalls and other measures. There is particular concern that network devices between source and destination, referred to as *middleboxes* [19], render communications impossible. This presents a dilemma for establishing VOs since this reduces the level of collaboration that is possible. Furthermore, many Grid applications assume that network connectivity is mostly unrestricted. Accordingly, VOs may face failed collaborations or considerable application reprogramming because the member sites have security policies that clash, resulting in more restrictions than anticipated in the design of their applications. Moreover, since the participants in a VO are independent of one another, their network security measures are likewise quite varied. These issues may be taken together as a *security interoperability problem* [7]. Somewhere in the interconnected networks involved, a mismatch occurs between what is allowed on one end and what is not allowed on the other. The result is a connectivity problem, where "benign applications cannot traverse a firewall" [21].

We have studied the likely scenarios resulting from security interoperability and explored or developed methods that facilitate connectivity despite such restrictions. The result is REMUS [25], a project in which we built a *re*routing and *mu*ltiplexing *s*ystem that enables cross-firewall connectivity while minimizing the need to modify or rebuild either the Grid applications or the firewalls. Grid-running software, including user applications and Grid infrastructure middleware, are encapsulated within a virtually connected environment with minimal or zero code modification. Likewise, rather than engage in firewall evasion and avoidance, REMUS complies with the security policies for a given network. Only authorized channels are used, and mechanisms are provided to minimize the impact on existing security measures. In addition, REMUS uses different connection mechanisms, depending on whichever is appropriate for a given scenario, e.g., using proxies or tunneling. REMUS is intended to support a library of such mechanisms, given that no single solution will always work. REMUS achieves "connectivity virtualization"—leaving Grid software and firewalls mostly or entirely untouched—by intercepting outgoing connections on all hosts and rerouting them across networks.

REMUS is also intended to be as portable as possible. Hence, our solutions are built from a number of standard secure communication protocols such as Secure Shell (SSH) [30] and SOCKS [15, 16]. In this manner, it becomes more likely for REMUS to be readily supported across different platforms.

This paper describes REMUS, which we designed to be a general-purpose Grid connectivity middleware that supports different Grid software and platforms. For purposes of testing and illustration, we implemented REMUS as a virtual connectivity middleware underneath the Globus toolkit and the Nimrod/G parametric modeling suite. However, the efficacy of our solutions for other Grid applications in similar situations will also be discussed. Given that network performance is affected by rerouting, we present performance measurements for various Grid operations with and without REMUS intervention.

This paper is structured thus: Section 2 discusses how Grid networking works and what firewalls are. Section 3 describes our case studies: Globus and Nimrod/G. Section 4 lays out the problems caused by multiple firewalls across Grids. We illustrate by discussing specific problems posed to Globus and Nimrod/G. Section 5 presents REMUS as a solution, while Section 6 discusses technical details about tunneling and port forwarding strategies in general. Section 7 discusses the implementation of REMUS in its current form, using SSH tunneling and SOCKS rerouting. We illustrate by discussing how it assists Globus and Nimrod/G, as particularly representative types of Grid middleware/applications. We also discuss security issues and features pertaining to REMUS. Section 8 describes our performance tests across two campuses of Monash University. We discuss related works in Section 9 and then conclude the paper in Section 10.

2 Grid Networking and Firewalls

Software running on Grids, which may be part of the Grid infrastructure or of a user-level application, can use the network in a number of ways. Software and utilities at the infrastructure level may use the network for job or data management transactions as well as Grid management tasks. Such operations typically follow a client-server model, where the client initiates a connection to the server and makes the appropriate request which the server complies with if all requirements are met. User applications may also involve network activities, including message-passing and data transfers.

For example, it is possible that a Grid application involves database operations. One of the components, C_{DB}, might be a database server listening for transaction requests. The client component, C_{CL}, might need to periodically send data down to C_{DB} through standard database transactions. Likewise, a user application may involve two components which share data, where one component must send periodic output to another component for processing. A workflow application will require that the output of one component is sent down to a downstream computation as the latter's input. Regardless of what sort of Grid software is using the network, firewall issues may arise where data communications of any type becomes impossible. In order to resolve such issues, three things must be understood properly: the nature of Grid networking, the nature of firewalls, and how software components on the Grid communicate.

2.1 A Grid Network Model

Figure 1 shows to the right the Grid architecture as described in a Grid anatomy by Foster et al. [8]. The Internet layered protocol architecture consisting of the Transport, Internet and Link layers is shown against a Grid architecture, but the correspondence needs to be clarified. There is no distinction here between connectivity for the Grid and connectivity at the fabric layer for non-Grid networking. We therefore present a two-layered stack formed by the *fabric connectivity* layer and a *Grid connectivity* layer above it. Note that the latter corresponds with the *connectivity* layer as

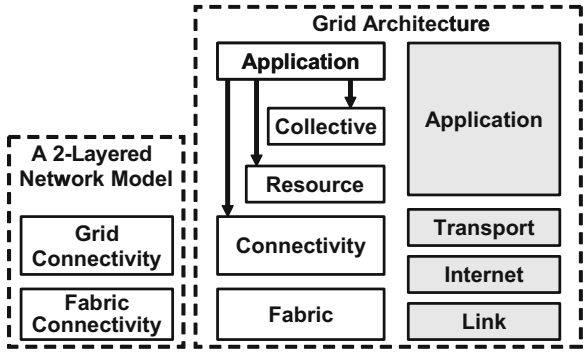


Fig. 1 The *two blocks at the left* form a two-layered model of Grid networking. The original Grid Architecture comes from "The Anatomy of the Grid" [8]

described in the Grid anatomy [8]. However, this two-layered network model highlights the difference between fabric-level networking and Grid-specific networking. Doing so allows us to distinguish between the nature of networking and firewall issues at each of those levels.

The bottommost fabric connectivity layer is the pre-existing communication infrastructure that interconnects the Grid fabric, typically designed and administered independently of the Grid infrastructure. Fabric connectivity is not managed on the Grid, although there is interest to merge the two levels of networking for purposes of optimization. The Grid *connectivity layer* involves Grid-specific functionality to support Grid protocols and standards, e.g., for authentication and authorization. For this reason, the illustration shows the Grid connectivity layer as overlapping the collective, resource and connectivity layers of the original Grid anatomy. We are not concerned at this point with a rigid mapping between network operations at the Grid connectivity layer and the corresponding operations at the fabric connectivity layer below. What matters is that we can separately analyse and deal with the firewall issues that come up on each layer.

Many organizations that operate Grid environments have underlying fabric networks that predate Grid computing, and Transmission Control Protocol (TCP)/Internet Protocol (IP) is probably the most preferred infrastructure [3]. TCP/IP is a protocol stack or suite that consists of the *transmission control protocol* and the *Internet protocol*

beneath it. Grid connectivity does not necessarily require TCP/IP implementations only, but its prevailing popularity for local area as well as wide area networking has made this the typical case. The Grid infrastructure is commonly a middleware layer on top of existing computing platforms, and TCP/IP is the common underlying network infrastructure. Our work is mostly confined within such implementations.

TCP/IP networking largely based on a *client/server* paradigm [3], with the basic notion of clients initiating the communications by sending a request for a service to a server, which, if appropriate, complies with the request. Much of the services offered over the Internet use TCP, which provides reliable, connection-oriented data transport between hosts that are identified by IP addresses. A different class of applications use another transport standard, the connectionless *user datagram protocol* (UDP), which does not guarantee delivery of data, but provides a simpler and faster mechanism [3]. Both TCP and UDP use the concept of *ports*, which are access channels, presumably available and accepting connections. A single Internet "host" machine may support many ports, and these ports may be bound to available services. By standardizing to a set of well-known port assignments to well-known services, it becomes easier to access services on the network.

We define a network domain D_i as being made up of nodes n_1, n_2,.., n_y, which are either service clients or service providers (or servers), interconnected over the network. A connection is made to each node using a certain protocol θ and a given destination port number$1 \leq p_k \leq 65535$. Assume for this discussion that we are only concerned with application-level protocols, e.g., SSH, TELNET, SMTP. We also assume that available ports are between 1 and 65,535. The parameters for a connection K are therefore the source address, the destination address, a source port number, a destination port number, and a protocol type. A connection over such a network may therefore be identified at any given time by a quintuple, $K = (n_i, n_j, p_k, p_l, \theta)$, which identifies both ends of a connection, by IP address and port number, as well as the protocol in use.

While Grid connectivity involves Grid-specific services, ports and protocols, they are still implemented on top of TCP/IP. Basic Grid services will therefore involve connections that are still identified via a quintuple $K = (n_i, n_j, p_k, p_l, \theta)$, where the protocol θ is Grid-specific but nonetheless based on TCP/IP. There are Grid-specific considerations involved concerning when a service may be granted. This is primarily about verifying the identity of the user behind a client request and the authorizations associated with that user for the requested services.

We may therefore distinguish between network usage at the Grid connectivity layer and at the fabric connectivity layer by determining if the protocols and ports involved are Grid-specific or not. This distinction is relevant in later distinguishing between security policies specific to Grid-related communication and those that are not.

2.2 Firewalls

Security policies implemented by firewalls will be typically segmented into logical partitions. A subnet within a larger network may be reserved for privileged users, whose access rights translate to lenient firewall restrictions. Policies may reflect departmental or institutional concerns, and so there may be departmental as well as institutional firewalls operating independently of each other. In the same way, security policies may concern fabric connectivity issues, which exist independently of Grid connectivity concerns. Those concerns do not always harmonize with each other. In some cases, a global or institution-wide security policy may conflict with a local or departmental security policy. In the same way, since the Grid reflects a policy of sharing, there may be conflicts between an institutional concern to limit access to resources and a local policy to open up resources for sharing on a Grid.

For example, all database access may be intended for authorized use only within the institution, but certain databases may subsequently be intended for sharing with authorized partners on the Grid. It is likely in this situation that all database servers are protected by the institutional firewall, which screens all externally initiated connections. This security policy is therefore implemented at the fabric level, while Grid-specific means of controlling or influencing accessibility

may not be available, i.e., the firewall is not a Grid resource that can be managed as such.

Another example that illustrates lost potential caused by firewalls is when applications on the Grid are physically allocated across different institutions. Communication across those networks may be harder for user applications because they do not use standard port assignments. A port is an access channel across a network, and port numbers for well-known services, e.g., telnet, SSH, FTP, are practically standardized. System-level software are likely to use those ports, and firewalls may admit access to those ports for standard services, including Grid services, e.g., job services and data transfers. While having firewalls open certain ports at the fabric level is not always welcome, opening up a few ports is less risky than opening up several. However, defining a range of several possible ports may be the only solution for user applications, whose port assignments may not be known until runtime.

Working towards Grid connectivity need not require changes that may compromise security policies at the fabric level. We contend that there may be ways to provide Grid connectivity even in these situations. To engineer a solution requires that we first analyze specific cases of fabric-level firewalls that block Grid connectivity.

Typical firewall restrictions operate by checking connection destinations against a list of open, closed and filtered ports. This applies to inbound, outbound, and forwarded connections, relative to the protected domain. Such firewalls are among the easiest to set up and among the most common. In many, if not most, operating systems, a firewall will be installed on the installation machine by default. If such a firewall was blocking all incoming TCP connections to port 5,432 on the database machine from outside the network, there is no chance to make a connection from outside.

Let us consider the parameters for such problems. We earlier gave a simple definition of a connection as a quintuple, $K = (n_i, n_j, p_k, p_l, \theta)$, which identifies both ends of a connection, by IP address and port number, as well as the protocol in use. Security policies regarding connections to a given node or its network translate to a finite "list" of allowed and disallowed connections. Note that there are different and independent lists of security restrictions prevailing for each firewall in a given VO. It is common to find several firewalls within the same organization, e.g., for each network or local firewalls for some nodes. Combining the various lists of the firewalls traversed makes for a potentially narrow window of accessibility. To simplify each situation between two nodes that need to connect, we focus on a single list which is a result of compounding the security policies across the two nodes and all networks between them. Our concern is in specific connections to port p_j on node n_B, i.e., $n_B.p_j$, from some source port on node n_A. This list includes ports that are simply open but not bound to any pre-existing service, or those that are through some specific protocol. Note that we use the term "connection" loosely in this paper. We do not necessarily imply a connection-oriented protocol such as TCP. This may also include UDP, which is connectionless. We say that a node n_A is connected to port p_i on node n_B via protocol θ when successful transport-level communication, e.g., TCP or UDP, begins and persists for the duration intended.

In a generic sense, whether at the fabric or Grid connectivity levels, successful connections require three things. First, there must not be any security restrictions in place that will block connections from n_A to n_B on port p_i. Second, the connection should have been initiated by a component c_A and accepted by component c_B. Finally, it is necessary that both c_A and c_B can speak the same language: protocol θ. The first requirement is generally understood to be a function of the network security policies, implemented by a firewall. The second requirement on component interaction is trivial: one component must initiate a connection that another component accepts since the point-to-point connection is existentially dependent on two peers. The last requirement is common sense: communication requires a communication protocol agreed upon by both parties.

2.3 Grid Software and Firewalls

In order to generalize the problem, we must be clear about the nature of Grid software communications and how they are affected by firewalls. Figure 2 shows a generalized model, where fire-

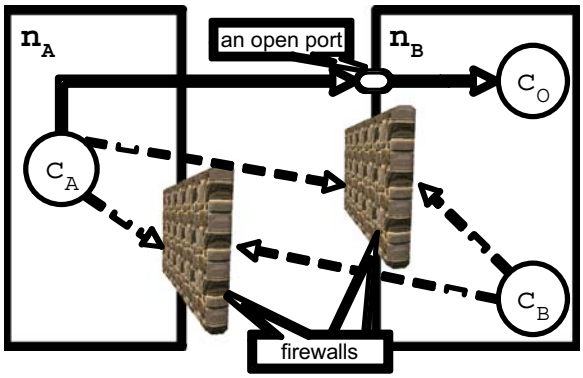


Fig. 2 The *four dashed arrows* show four different paths that may be blocked by firewalls when C_A and C_B try to communicate using sockets. In contrast, C_O is accessible via an open port which is never blocked by a firewall

walls situated on two sites may each block outbound and inbound connections between C_A and C_B. The exception is C_O, which listens behind an open port that is not blocked by a firewall. The situation involving the former two illustrates a worst-case scenario, where direct socket connections are blocked in both directions each time. REMUS was designed so that such situations can be dealt with in various ways. We illustrate this by investigating the case of Nimrod/G [1, 2] and Globus Toolkit [9] in such a scenario, where REMUS must assist in outbound connections as well as inbound connections for both sites. Nimrod/G is particularly apt since Nimrod agents must be pushed out from the Nimrod/G site (outbound) to computational nodes, which must accept the launch request (inbound). Once executing, the agents must then contact the Nimrod/G site from the computational node, and Nimrod services must be accessible (inbound) so that the agents may pull down experiments and experiment data. It is thus sufficient to illustrate REMUS using Nimrod/G since it provides us with the worst case to investigate. But what about UDP transports? The above scenarios do sound more applicable to TCP streams, which are connection-oriented. UDP traffic is connectionless, but they are treated in almost the same way by firewalls. Most Internet services are TCP-based, so that firewalls are more likely to restrict TCP connections. On the other hand, some services are accessible using UDP, and these may also be restricted by firewalls. Such restrictions take the same approach, identifying ports and IP addresses that are to be either accessible or blocked. REMUS is designed to also redirect UDP traffic, since the rerouting mechanisms we use are able to tunnel UDP traffic. However, we did not include such tests in this paper since, in our experience, most Grid-based transactions are carried over TCP.

3 The Globus Toolkit and Nimrod/G

This section briefly introduces the Globus Toolkit [9] and the Nimrod/G suite of tools [1]. Both are used as case studies in order to demonstrate that typical Grid connectivity can be facilitated through REMUS. Note that, while much of what we will discuss would apply to several versions of the Globus Toolkit, we only used version 2.4 (GT2.4) for our experiments. We provide some discussion about newer versions of the Globus Toolkit in the next section. These versions seem at first glance to be less affected by firewall issues, but the situation is actually unchanged. As for Nimrod/G, much of what we present will be specific to Nimrod/G 3.0 agents over GT2.4 resources, but again, the situation is no different from the later versions of Nimrod/G, regardless of the Globus Toolkit version underneath.

3.1 Case Study 1: The Globus Toolkit

The Globus Toolkit [11] provides various client and server tools to manage Grid resources. It has become a de facto standard in building Grid testbeds, thus providing us a good case study opportunity. Figure 1 illustrates how the Globus toolkit provides the connectivity, collective and resource layers of the Grid. It facilitates operations over underlying resources, i.e., the Grid *fabric*, and tools for job management and data management. Globus typically deals with a pre-existing resource manager, which manages several physical resources as one resource on the Grid, e.g., a queue manager for an entire cluster. Connectivity with and among Grid resources is supported by several protocols in order to build connections, authenticate users, authorize

access and delegate tasks. Security-related connectivity is provided within a *Grid Security Infrastructure* (GSI) which is based on public-key cryptography. Grid resources, which may be abstractions over the fabric, may be accessed using a number of protocols, including the HTTP-based *Grid Resource Access and Management* (GRAM) protocol and GridFTP for file transfers. GRAM controls the allocation of resources' compute nodes and the monitoring of computations allocated to them. GridFTP is an extension over the standard File Transfer Protocol (FTP) with GSI mechanisms and enhancements that include the ability to use multiple data streams. The two server-side components that facilitate GRAM and GridFTP are the *Gatekeeper* and the *GridFTP server*. The Gatekeeper accepts submissions for computational jobs and allocates them to the appropriate *job managers*. In some cases, job managers will deal with autonomous resource managers, rather than directly with the resources themselves. For example, some job managers will deal with particular types of batch submission systems. The job manager will also return status data and the job output to the GRAM client. The GridFTP server provides data transfer mechanisms, facilitating the transport of files to or from some remote source. GridFTP supports multiple parallel streams to maximize throughput, and operates within the GSI framework for security purposes.

Figure 3 illustrates some of the Globus Toolkit components in a snapshot of a running system. In this simplified scenario, a Globus client may either transact via job or file transfer operations. The Gatekeeper receives job-related requests and spawns off an appropriate job manager for that operation. While the initial traffic between the client and the Gatekeeper is in one direction only, traffic between the client and the job manager may be initiated in both directions. Data transfer operations are delegated to a spawned FTP server, one for each transfer request. This FTP server may open several ports, depending on how many parallel streams are requested, then data streams proceed in one direction, always initiated by the Globus client. Collectively, all Globus transactions will require that connections can be initiated in both directions.

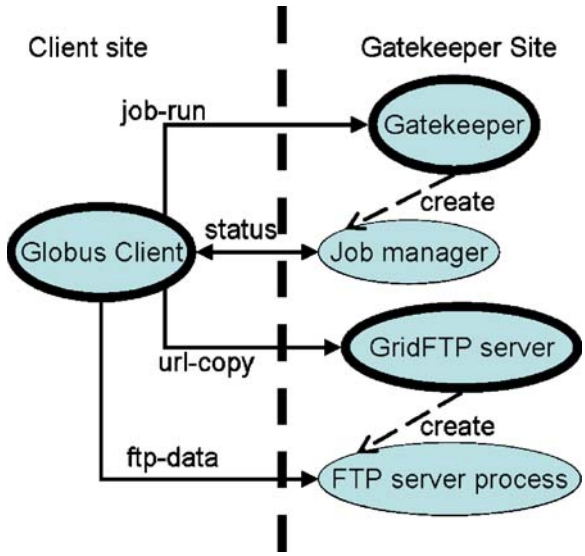


Fig. 3 This is a snapshot of Globus Toolkit components that communicate across site or network boundaries. A firewall that stands in the middle can easily block normal connectivity

The Globus toolkit firewall requirements document [28] defines certain access settings that are necessary to facilitate successful Globus transactions. Apart from the service ports for the Gatekeeper and GridFTP, several ephemeral ports (above port number 1024) must be accessible for incoming and outgoing TCP traffic. Note that the Gatekeeper and the job manager, from the Gatekeeper site, may initiate connections back to the client site. This *callback* functionality may require client sites to permit incoming traffic to up to 10 ports for each expected simultaneous user. At the server site, 20 ports are recommended [28]. For data transfers, multiple connections will also be initiated, from the GridFTP server, back to the client site. Having several new TCP streams that follow the initial one, proceeding from both directions, makes firewalls more difficult to handle.

3.2 Case Study 2: Nimrod/G

We look at Nimrod/G as a good case study of a Grid application. It is typical of applications that execute jobs on Grid resources using standard job and data transfer operations. **Nimrod** is middleware that sits above and harnesses the computa-

tional power of several loosely-coupled processors to execute experiments [1]. Nimrod experiments are *parameter sweep applications*, involving tasks that process a large range of parameter combinations. Areas where parametric applications are useful include design and engineering, as both areas benefit from the simulation of their target product over several different situations to study the effects of changing parameters, e.g., weather and environmental conditions, energy characteristics, load, etc. [1, 24].

Nimrod/G is the Grid-enabled version of Nimrod, consisting of client components, server components and computational components. Client components include command-line or web browser interfaces as well as the tools and libraries they need to define and manage experiments. Server components are at the core of managing parameter sweep applications, including the scheduler, a centralized database server and a *Nimrod master* to manage the other Nimrod components and the interaction with Grid resources. Computational components run on resources to manage the execution of jobs as required by Nimrod experiments.

When an experiment is started up, an *experiment master* is created, which will invoke a new *file server* as well as a *database server* if they don't already exist. The file server handles file transfers while the database server handles control data for executing jobs. The appropriate *actuator* for the chosen Grid resource must then submit the correct agent binary as a job to be executed. This is determined by the resource's architecture, kernel and operating environment version. If the binary already exists on the resource, it is simply reused. When it starts up, it will connect to the Nimrod central database to verify its "marching orders", e.g., jobs to execute, input files to download, and whether or not it runs as a *Nimrod Agent* or as a *Nimrod Proxy* on the head node. The task of an agent is to run the experiment using an assigned parameter set. By farming out several agents across Grid resources, Nimrod/G can parallelize the parameter sweep and reduce the overall execution time across the entire range of parameter sets, each processed as manageable tasks. The Nimrod Proxy is, as the name implies, a proxy utility to centralize the communication between the agents and the file and database servers. The agents, directly or through the Nimrod Proxy, talk to both the database server and the file server throughout the experiment until the results are sent back. This necessitates a connection from the agent to two distinct ports behind which the two Nimrod servers will listen [1]. Figure 4 illustrates a snapshot of Nimrod/G components at work as they manage an experiment. The dashed line running down the middle signifies network boundaries. There are client components and server components. Note that Nimrod job submissions and data transfers

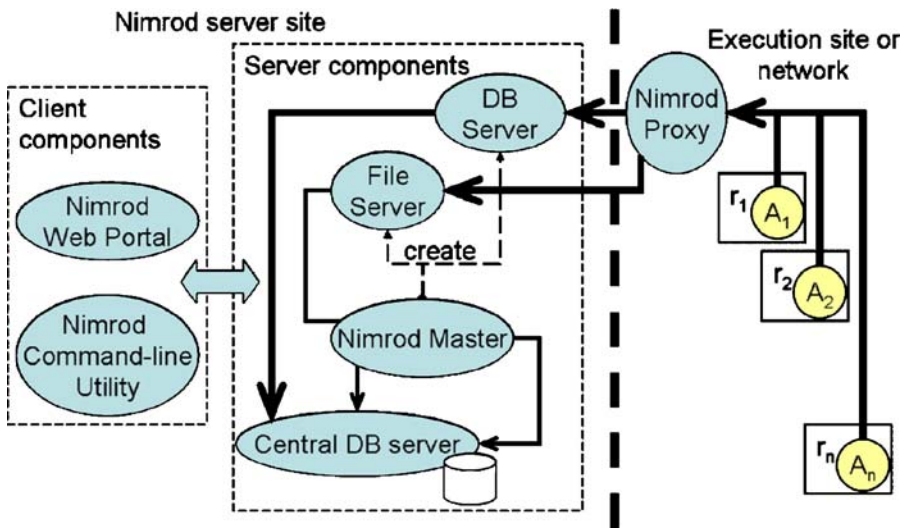


Fig. 4 This is a snapshot of Nimrod/G running across two sites. The interconnection is necessary for Nimrod to manage the agents (A_1 through A_n) on remote compute resources (r_1 through r_n), which in turn manage computational jobs and data transfers that they might require

via Globus will be treated as transactions with a Globus client. For example, Nimrod submits experiments as jobs using globusrun, which is a Globus client utility. However, as far as Nimrod is concerned, the server components of Nimrod are service providers. To the left of these components in the diagram are client components. At present, we are actively using two: the portal that can be accessed via web browsers, or a command-line utility which provides the same client functions.

Communication between the Nimrod agents (on the right) and the file and database servers (on the left) go through a Nimrod Proxy in this particular scenario. Connections from the Nimrod Proxy, or from the agents if the proxy is not deployed, are all outbound, initiated by the agents. Interconnectivity among Nimrod components across a Grid testbed must be bidirectional. The initial connections to Grid resources to deploy or invoke the Nimrod agents and proxy require outgoing traffic initiated by Nimrod as Globus GRAM submissions. During the processing of an experiment, Nimrod agents must be able to make outbound connections back to the Nimrod database and file servers, for the effective management of experiment tasks.

4 The Problem with Firewalls

The primary purpose of firewalls is to protect resources or networks by blocking particular incoming or outgoing communication across the network. This presents us with a conflict between the barriers that firewalls must enforce and the accessibility which Grid middleware take for granted.

4.1 A Classification of Conflicts

In an earlier work, we considered three cases of hindered communications between two components communicating via the network [25]. A *node conflict* occurs when security restrictions block connections to the target node at any port. A *port conflict* occurs when security restrictions block connections to a particular target port only. We may also find that, while a port conflict exists, there is instead another vacant port available. In such a situation, the connection still cannot be made to the intended component until it binds to that vacant port or use some bridging mechanism that will facilitate communication. We can also encounter a *protocol conflict*, when a successful connection can be made to the node at a given port but the service listening behind that port is unable to communicate using the protocol expected by the connecting component. Grid middleware like the Globus Toolkit and Nimrod/G are dependent on the network. Both assume that it is possible for various components running across remote machines to interact across the network using multiple ports and protocols. Any of the conflicts mentioned above can disrupt network activities of the Globus Toolkit, Nimrod/G, and Grid applications in general.

4.2 Firewalls and the Globus Toolkit

While it is possible, firewalls are usually not situated between Globus toolkit components and computational resources. However, firewalls can block connections in one or the other direction between the Globus client requesting for services and the Globus service components waiting for requests or processing them. Certain ports must be accessible through the firewall in order for the toolkit to function properly [28]. To begin with, ports to access Globus services must be accessible, including those for the Gatekeeper, the GridFTP server, and additional ports that are required at runtime, which can be set via the GLOBUS_TCP_PORT_RANGE variable. If connections to these ports are unsuccessful, GRAM and GridFTP operations cannot proceed. The GRAM service may also use *callbacks* to the client for status notification. Unless incoming connections to the clients are allowed within a certain port range, GRAM callbacks will fail. GridFTP has its own unique requirements. For the sake of performance, the Globus toolkit uses many ports for parallel streaming. However, in a secure environment, the number of open ports is deliberately kept at a minimum, which would rule out the port range configuration.

Even for the newer versions of the Globus Toolkit, which have been available for a few years now, the problem with firewalls has not

disappeared. In the newest configuration guides describing how to deal with firewalls, the same limitations prevail [28]. Much of the changes in the GT3 and GT4 pertaining to connectivity simply deal with adopting web service standards. Hence, job initiation and management use web services on new port assignments, but requirements for multiple ephemeral ports remain. For example, event notification and file staging still proceed from server-side ports to client-side ports. File transfers using GridFTP do not change at all. The bottom line is that a fair number of ports must still be directly accessible on both sides. Approximately 10 ports must be accessible at the client site for each simultaneous user, and up to 20 ports must be accessible at the server side for each simultaneous user. This remains undesirable where the client and server sites are required to maintain strict network security policies.

4.3 Firewalls and Nimrod/G

For Nimrod/G, firewalls can get in the way between the computational resources and the Nimrod server components (see Fig. 3). When Nimrod/G uses a Grid resource for the first time, it must first transport a copy of the Nimrod remote binary to the computational resource. This binary encapsulates both the Nimrod agent and the Nimrod Proxy. If the firewall blocks the transfer, then Nimrod may not proceed with using the resource at all. If the binary is present, submitting experiments is a matter of initializing the agents on a given computational node for a particular task over a specific parameter set. This invocation is usually through a normal request to execute a job, e.g., through a queued submission. The firewall can get in the way and Nimrod will not be able to use the resource if this request is not received by the intended job manager. Assuming it is initiated, the agent must connect back to Nimrod server components to obtain control and input data. The agent must talk to both a database and file server, so it must have access to two distinct server-side ports on the Nimrod site where the two servers accept requests. The firewall protecting the computational resources may block these connections, making it impossible for the agents to receive their tasks for processing. We may deploy a Nimrod Proxy to centralize all connections from the agents to the Nimrod servers, but the firewall may likewise block connections from the Nimrod Proxy. It could be deployed elsewhere, but the agents must be able to reach it and it must be able to reach the Nimrod server components. Furthermore, during the processing of the task assigned to it, the agent sends "heartbeats" to Nimrod, "I'm alive" status updates which the Nimrod master must keep track of in order to manage the active agents. If the heartbeats are blocked by firewalls and are not received by the Nimrod server components, the agents are assumed to have failed, and their jobs must be restarted. Finally, assuming that the experiment is completed, the agent must be able to transmit its output back and perhaps receive additional tasks to process.

5 Rerouting Via REMUS Using Wrappers

We earlier mentioned our classification of firewall issues as port, node or protocol conflicts. We previously addressed some of these problems with ad hoc solutions based on what channels were allowed, using whatever mechanism was appropriate [25], However, the philosophy behind REMUS is not tied down to any single firewall traversal mechanism. REMUS advocates a multipronged, unobtrusive, rerouting, middleware solution: In our design, Grid applications should have access to *rerouters*, wrappers that will intercept normal communications and get them across to the intended party using whatever means available. Several traversal mechanisms are appropriate at different situations, and REMUS should be capable of as many of them as possible. Figure 5 illustrates the abstract model by which rerouters bridge connections between components that seek to communicate despite the presence of firewalls between them. Two communicating components C_A and C_B are separated by one or more firewalls that may restrict direct communications between them. Rather than force the reprogramming of the components or the firewalls, we advocate a wrap-and-reroute approach. Two rerouters ρ_A and ρ_B are shown to be wrapped around each communicating component. All network connections will be intercepted and rerouted as appropriate. There

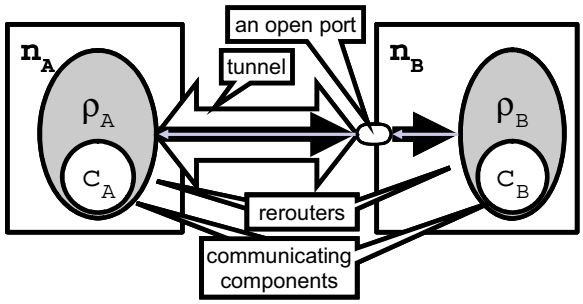


Fig. 5 The abstract model of REMUS illustrates the two central ideas of diverting connections unobtrusively via wrappers and rerouting

is a primary tunnel shown as a bi-directional block arrow between the two nodes. This tunnel will cross several nodes, networks and firewalls, if necessary, and may involve several proxy hops. The tunnel may also feature compression and security features, depending on what is required or available. Forward and reverse connections, shown as black and pale arrows in opposite directions, are connections between the components that are carried over the rerouting tunnel.

The basic REMUS architecture addresses the classification of conflicts earlier mentioned in a straightforward manner. In a vacant port situation, since direct connections to the vacant port are possible, then we position rerouters on the vacant ports as required. If there is a port conflict with an existing daemon, then rerouters will be deployed, if possible, as plug-in or extension components for the pre-existing daemon. For example, we may deploy a web service implementation of a rerouter. In a node conflict situation, where one or both nodes cannot be accessed via any port, we can use a proxy approach where rerouters must work with one or more rerouters sitting on mutually accessible third-party proxy nodes. If necessary, a proxy chain is formed with however many hops are required to establish the tunnel. Rerouting is not possible in cases where the firewall restricts connections to or from one of the nodes completely. This is true not only for REMUS but for any solution that can be devised. Tunneling is not possible unless one port is at least accessible for establishing the tunnel. In such a situation, the only recourse is to lift restrictions on the firewalls or middleboxes where they are implemented, i.e., opening *pinholes* [21].

6 Tunneling and Port Forwarding Techniques

Firewalls and similar middleboxes have become commonplace appliances on the Internet and within organizational networks. The restrictions they impose have prompted people, for different reasons, to seek various means of circumventing the restrictions, either openly or covertly. While the motivation behind such activities is at times questionable, there can be legitimate reasons for resorting to them. It is possible that a group of users, with authorization, wishes to explore an application, middleware or operating environment which requires network access that is not otherwise permitted by the firewalls already in place. It may also be a preference to leave the firewall configured as it is, and institute an independent access mechanism that may be more controllable and flexible than the firewall. Security policies implemented by firewalls are considered to be part of the infrastructure, and applications seeking to use the network legitimately for authorized purposes may simply have to work with them. Here we mention some of the techniques and tools that have been devised to work around firewalls.

SSH is commonly used for tunneling, and has been recommended for Globus firewall issues [12, 28]. Many secure networks allow SSH traffic through because some form of login must be tolerated and SSH is highly secure. With pure SSH tunnels, the assumption is that the SSH service on the remote site is available and accessible. It is then possible to program the SSH client to use several pairs of mapped ports. The local port is set up to listen for connections, which are then forwarded to the counterpart remote port on the remote destination host.

SSH may also piggyback through another transport mechanism if direct SSH connections are not possible. One that we tried was using the HTTP proxy or web proxy service to deliver SSH connections, assuming that the service was available and accessible. Proxies allow traffic to go physically around firewalls, by rerouting connections through a third party that is accessible as well

as authorized to send network traffic out across the firewall.

Whether or not we use SSH directly or via a proxy, port forwarding can only assist in getting the connection across in one way. Problems are encountered as connections are made in the other direction over multiple ports. It is not entirely possible to predict which ports will or should be used by new connections. The GLOBUS_TCP_PORT_RANGE is an environment variable that allows the Globus toolkit to restrict whichever ports *can* be used within that range, but that does not provide enough control. We may designate 1-to-1 port maps between several local ports and several target remote ports, but there is no way to determine which of the ports (and thus, its counterpart) will be used by a particular instance of a Globus or Nimrod/G component.

For example, when the GRAM client executes via globus-job-run, it will listen on specific ports, e.g., 65502, 65503. We assume that, due to the firewall, those ports cannot be accessed directly. Instead, the job manager will have to rely on a tunnel. For example, we can run the SSH tunnel so that ports 14000 through 14535 on the job manager site will map to ports 65000 through 65535 on the GRAM client site. The job manager must therefore use the local ports 14402 and 14403 for callbacks to reach the right target ports, 65502 and 65503, on the GRAM client site. However, the job manager only doesn't know about which local ports map to the right ports on the GRAM client site. After all, it only knows about those ports (65502 and 65503) because the GRAM client told it so, and the latter does not know about the port forwarding at all. Some other mechanism must set up the right port to map for the job manager, e.g., map local port 35502 to remote port 65502, and so on. One way to get around this problem is by submitting batch transactions, and results are cached until claimed. In this manner, it becomes unnecessary for the job manager to connect back to the GRAM client to deliver results or status information. The port through which the results can be obtained is known via a GASS-cache URL, e.g., https://globus.csse.monash.edu.au:4102/12357/1153210197/, but the given port, 4102, cannot be reached behind the firewall. The request to access it must go through port forwarding, with an explicit URL that reflects the local port assignment [12]. This involves having to determine that URL after the batch submission is made, since the ports used by Globus are dynamically allocated, as are the GASS-cache URL strings. This illustrates a need for a flexible tunneling mechanism to hide the details of port forwarding.

The situation is slightly different with Nimrod. We must assume that Nimrod agents are deployed successfully via job submissions that go in the resource-bound direction. The port forwarding allocation of local and remote ports may then be predetermined via statically assigning the port forwarding, e.g., the local ports X and Y forward to the database and file server ports of Nimrod, respectively. On the other hand, with several agents running on multiple resources, it would be inefficient to create tunnels for each of them. Centralizing their connections back to Nimrod makes sense, through a Nimrod Proxy, which can sit on the front node, but the hosting node dynamically allocates the ports that will be allocated to the proxy. Normally, the Nimrod Master only learns of those ports when the Nimrod Proxy reports to Nimrod after successfully starting up. The Nimrod Master may then invoke the agents and include the Nimrod Proxy contact details when they are started up. Setting up the Nimrod Proxy on a static pair of ports reduces its flexibility, which comes from its ability to run as a normal job allocated by the hosting resource manager. For example, it is possible to bridge the gap between remote agents and the Nimrod server components through multiple proxies in a chain. In that scenario, each Nimrod Proxy sets up listening ports. Agents can use these ports directly or through other Nimrod Proxies, which can be deployed for intermediary roles as the need arises. It is always possible to "hack" a solution to the firewall issues described by patching up mechanisms here and there, as the situation allows, with the intervention of systems administrators. Grid application components may be rewritten slightly and firewalls may be altered as the need arises. It is far more desirable, however, to engineer a solution that abstracts the complexities of firewalls, tunnels, ports and protocols. For this reason, we developed REMUS to

be unobtrusive and geared towards using standard communication mechanisms through channels that are authorized by existing security policies.

7 The Implementation of REMUS

Figure 5 illustrates the principles behind REMUS. For one, the mechanism that provides rerouted transport is not fixed. Tunnels, after all, can be constructed in several different ways, e.g., IP over TCP, TCP over TCP, or UDP over TCP. In cases where the tunnel is transported using TCP or UDP, the protocol carrying the tunnel may also vary, e.g., SOCKS, SSH, web proxy. Whichever one is actually in use will depend on what the situation allows. In this section, we will provide details concerning the implementation of REMUS that assumes the typical, restrictive scenario:

- We have no direct connectivity to the Grid service ports (2811/tcp, 2119/tcp, 8080/tcp or 8443/tcp for Globus),
- nor to the ephemeral ports that the Grid software requires,
- but SSH is available.

We therefore create tunnels using SSH, and use SOCKS in order to invisibly reroute and multiplex connections across. This is not intended to be the only implementation of REMUS, but we consider this to be practical in this likely scenario where tunneling is still possible through SSH. It has the added benefit of encrypted transport for authorized users with explicitly granted accounts.

As explained in Section 6, it is undesirable to rely on hard-coded maps between the local port and the remote destination port, since the remote destination port cannot be pre-set. We therefore prefer a dynamic port-forwarding tactic, which supports multiple ports that are not predetermined until the ports are accessed. We rely on proxy-based solutions because direct connections are often blocked, so we prepare for the worst case that remains workable. We expect to use a third party proxy node that is mutually accessible to the communicating hosts. We have tested proxy solutions that use web proxy and SOCKS proxy methods [12, 16]. The web proxy creates the connection on behalf of our rerouter, which sets up an SSH tunnel via the CONNECT command. However, web proxy servers often accommodate connections to port 80 (HTTP) or 443 (HTTPS) only, whereas we must create the SSH tunnel via port 22/tcp, Globus service connections via port 2119 and 2811, plus the ephemeral ports within the Globus TCP port range. A separate SOCKS proxy is therefore better since it is not restricted to only one target port, as it is intended to be a more generic proxy protocol for TCP and UDP [15, 16].

However, there are limitations. Institutions do not always provide either a web or SOCKS proxy for *incoming* connections. It is more common to find that only the SSH port, 22/tcp, is accessible on secure networks. For this reason, we favor the use of SSH as the underlying tunnel transport. On top of that, we still needed a way to intercept and reroute network traffic to use the SSH tunnels. For rerouting, we decided to use the open source *Dante* implementation of SOCKS [4]. It allows connections to be wrapped transparently within an environment that puts SOCKS functions ahead of normal network system functions. This approach results in transparently rerouted connections via SOCKS tunneled under SSH, with minimal reconfiguration of firewalls and Grid applications, if at all. This implementation with SSH and SOCKS, presents a clear and straightforward illustration of the wrap-and-reroute approach.

7.1 Setup and Configuration

Setting up the SSH + SOCKS implementation of REMUS is straightforward, even in the absence of an automatic or wizard-like deployment and configuration interface:

1. There is an assumption that user accounts are available on both networks, by which an SSH session may be established in both directions. This may be through a normal SSH connection with passwords, or with public/private key-pairs, or with X.509 certificates using OpenSSH with GSSAPI support [14].
2. The SSH connection is established with the preferred encryption and compression

settings, but dynamic port forwarding must be enabled. For example:

```
$ ssh -D 1080 username@remotenet.org
```

The switch "`-D 1080`" enabled SOCKS proxy services via the local port 1080. Since encryption and compression settings were omitted, then this session will use default settings. Settings which we recommend are "`-o 'CompressionLevel 9'`" for maximum compression and "`-c blowfish`" to use the fast blowfish encryption algorithm. Other settings are possible.

Note that another SSH connection is required from the opposite direction if connections are also not possible otherwise. The assumption is the same: we assume that SSH is allowed through this direction and the necessary user account is available.

3. Dante must be available on both ends to provide SOCKS client features in order to reroute outgoing connections. A SOCKS client configuration file (socks.conf) will determine which outgoing connections may use the SOCKS connections. The rules express, in a straightforward fashion, whether an outgoing connection to a given IP addresses and port will be facilitated through the tunnel, using which SOCKS proxy service. The appropriate proxy service associated with each rule is identified also by IP address and port number. Here is an example of a simple rule to forward connections bound for the subnet 172.17.171.0/28 via a SOCKS proxy running locally (IP address 127.0.0.1) from port 1080:

```
route {
 from: 0.0.0.0/0 to: 172.16.171.0/28
  via: 127.0.0.1 port = 1080
 protocol: tcp udp
 proxyprotocol: socks_v4
 method: none
}
```

Note that there are several possible options in how the configuration may be given. For example, whether SOCKS v4 or v5 are supported by the SOCKS proxy indicated, or whether connections are authenticated by username.

4. The SSH clients on both ends provide outgoing SOCKS proxy services via the dynamic port forwarding port specified at invocation. This is the simplest approach, but a more secure solution is possible. Instead of using the SSH client to provide proxy services, a remote SOCKS daemon via Dante may instead be used. The Dante SOCKS daemon has its own configuration file (`sockd.conf`) with rules that determine which incoming connections will be permitted. Using a full SOCKS daemon such as Dante has the advantage of effectively putting an inner firewall at the end of SOCKS tunnels. It is only necessary that the SOCKS client be configured to point to the Dante SOCKS daemon instead of the SSH client. Here is an example of configuration rules for the Dante SOCKS daemon:

```
logoutput: stderr
internal: eth0 port = 1080
external: eth0
method: username none #rfc931
user.notprivileged: nobody
user.libwrap: nobody
client pass {
 from: 127.0.0.1/32 port 1-65535
  to: 0.0.0.0/0
}
client pass {
 from: 127.0.0.0/8 port 1-65535
  to: 0.0.0.0/0
}
client block {
  from: 0.0.0.0/0 to: 0.0.0.0/0
  log: connect error
}
pass {
 from: 127.0.0.1/32 to: 172.16.171.0/28
 protocol: tcp udp
}
block {
  from: 0.0.0.0/0 to: 0.0.0.0/0
  log: connect error
}
```

This configuration allows local connections only, which makes sense if the SOCKS client connects through an SSH tunnel from a

remote site. This setup is ideal to give local administrative control over the SOCKS connections. Note that the final "pass" socks-rule permits connections intended for nodes in subnet 172.16.171.0/28, which we may assume to be a network of Grid resources. The SOCKS daemon must therefore have network access to those resources. The daemon itself may be situated wherever preferred, as long as the clients can access it either directly or indirectly.

5. Grid components must be invoked in such a way that the SOCKS libraries of Dante are preloaded. This is how network connections are implemented using SOCKS rather than as normal socket connections. In Linux implementations, this is done by appending the SOCKS library file, e.g., `/usr/lib/libsocks.so`, to the environment variable LD_PRELOAD.

   For convenience, we used shell scripts that initiate the SSH tunnels and generate the SOCKS configuration file:

```
#!/bin/bash
LOCAL_SOCKS_PORT = $1
SSH_USER = $2
SSH_SERVER = $3
SSH_CLIENT = /usr/bin/ssh
#
## Loop will force a restoration
of the tunnel in case it drops
out
#
while [1]; do
#
## High compression, fash
cipher, and set up SOCKS proxy
at requested port
 ssh -N -o 'CompressionLevel 9'
-c blowfish -D $LOCAL_SOCKS_PORT
\ SSH_USER@$SSH_SERVER
done
```

   These scripts, which execute under the bourne-again shell environment (bash), are straightforward and quite trivial. The creation of an SSH tunnel simply requires the preferred SOCKS proxy port as input. This same port number is input to the SOCKS configuration script, given that it must specify which local port to use for a specific destination. Typically, this is a local port, so that it is only accessible from within this same host. In other situations, it may be a port on another machine that is always accessible, e.g., an officially designated proxy server. In our tests, we only had one local destination for each SOCKS configuration, since we only involved two Globus hosts in our tests, one acting as a client, and the other as a server, and one proxy for all connections in that direction (client to server). In advanced configurations, the scripts would accept a range of port numbers, and would create several SSH tunnels and configure SOCKS to use several SOCKS proxies. The scripts are the same, either way, differing only in the range that we supply. Since we used OpenSSH without GSSAPI in our tests, it was necessary to first launch and authenticate the SSH agent, allowing us to make subsequent SSH connections after having only supplied the passphrase once. Certificate-based SSH access is not standard for all SSH-capable servers, as it requires specific installation/configuration of SSH services. If it were available, then the once-off authentication via the SSH agent is replaced by the generation of one proxy certificate which may be valid for several hours or days. In generating one proxy certificate, authentication via a passphrase is also required once.

7.2 Globus and REMUS

We explored the efficacy of REMUS via SOCKS on the Grid by applying it to the Globus Toolkit and Nimrod/G. At the core of Globus transactions are job management and data transfer operations. We focused on how to support these operations despite the restrictions imposed by firewalls. Nimrod/G needed a slightly different solution. Nimrod agents invoked by the job manager on the compute nodes are network-aware and Grid-aware components that are responsible for pulling in computational tasks and data to process as part of a parameterized experiment. As with Globus components, firewalls will hinder the

interconnections between Nimrod components, particularly where the agents must interact with Nimrod server processes. While it is possible to reprogram Grid applications in order to support rerouting across firewalls, there are considerable advantages in using the unobtrusive middleware approach.

Figure 6 gives an overview of how REMUS assists Globus Toolkit 2.4 (GT2). There are actually two levels of connectivity involved. First, there is the Globus connectivity level for Globus components. These connections are via TCP and typically use two service ports for connecting to the Gatekeeper and GridFTP services and several ephemeral ports within a certain port range. The latter ports are used on both the client and server hosts to restrict the ports that need to be open for incoming and outgoing connections. In order to facilitate the connections across firewalls, we must construct a second level of connectivity underneath, REMUS connectivity, which reroutes everything through a tunnel while remaining almost invisible to the Globus components. The rerouters invisibly come into effect through environment variables that need to be set and configuration files for the Gatekeeper and GridFTP servers that must include directives for those variables. The actual communication protocol to be used depends on the security policies in place, but we also considered which protocols can do flexible port rerouting. Our reference implementation of this technique was to use SOCKS wrappers on top of (or within) SSH tunneling. Figure 6 illustrates the SSH tunnel as a side arrow that spans across from SITE 1 to the DMZ host on SITE 2. Note that connections to the Gatekeeper go through this SSH tunnel. The assumption is that SSH sessions are allowed between the sites involved. OpenSSH supports SOCKS as a proxy protocol, and SOCKS wrapping is implemented by "socksified" connections: all communication between components are transparently coursed through SOCKS instead of directly via normal TCP. The technique calls for components to load SOCKS libraries into memory, making the components operate within a socksified environment. This includes job managers and binaries invoked on behalf of the user by the job managers. The SOCKS-enabled behaviour is therefore inherited from the gatekeeper down to the bottom-most computation. At the client side, the same preparations make sure that Globus client binaries are likewise socksified. The result of this virtual environment is that all Globus components communicate via SOCKS without being explicitly invoked with hard-coded tunnel settings.

SOCKS connections are carried through the SSH tunnel connected to an authorized host, which is typically located at the demilitarized zone (DMZ) of one or both networks. The assumption is that the network security policy permits incoming SSH connections directly to the Globus server components or to the DMZ host that can access them. SOCKS proxy services are provided

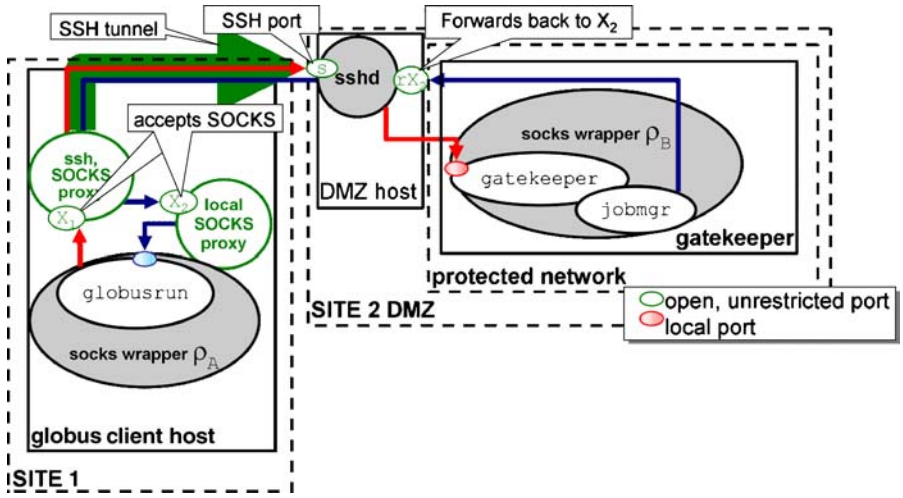


Fig. 6 Globus connections are enabled across firewall restrictions via SOCKS wrappers. The SOCKS connections are in turn tunnelled through a single SSH session with port forwarding

by SSH, a feature included in OpenSSH [29]. However, since OpenSSH SOCKS connections only go in one direction, we needed two SOCKS proxy mechanisms. We created a second SSH session which tunnels SOCKS back the other way, although we can also use reverse port forwarding through the first tunnel to send connections back to where a local SOCKS proxy is running, as shown in Fig. 6 (see connection via port rX_2 to the local SOCKS proxy component on `SITE 1`).

7.3 Nimrod/G and REMUS

Nimrod/G behaves almost like a typical Globus client for Globus transactions, and uses Globus client binaries. Through SOCKS-wrapping, Globus transactions between Nimrod/G binaries and the resources running from the Globus resources are enabled across the SSH tunnel. However, instead of simply submitting computational jobs, Nimrod/G talks to networked components: Nimrod agents. These agents must be able to connect back to the Nimrod server components, perhaps past another set of firewalls. A proper configuration of the Globus job manager provides the SOCKS-enabled environment to the Nimrod agents when they are executed. Figure 7 shows how SOCKS wrappers around the Nimrod agents or proxy allow connections back to Nimrod/G File and DB servers (FS and DBS). The connection goes via reverse-direction port forwarding using port rX_2 on the DMZ host, through to a local SOCKS proxy on the Nimrod/G host.

The implementation and configurations described were enough to enable GRAM transactions, GridFTP copies and Nimrod/G experiments to successfully run over two networks separated by firewalls that restrict connections in both directions. There is a reasonable trade-off between total connectivity failure and optimal network performance. To illustrate, we ran some GridFTP experiments.

7.4 REMUS and Security

REMUS was designed in order to transport Grid data exchange across firewalls. Since firewalls are typically implementations of a given organization's security policies, then any ability to escape firewall restrictions must be considered carefully. It cannot be denied that REMUS presents a lowering of network security, but it is designed to do so only for purposes of Grid computing. REMUS is designed to be part of the Grid infrastructure. It must be administered as an application layer

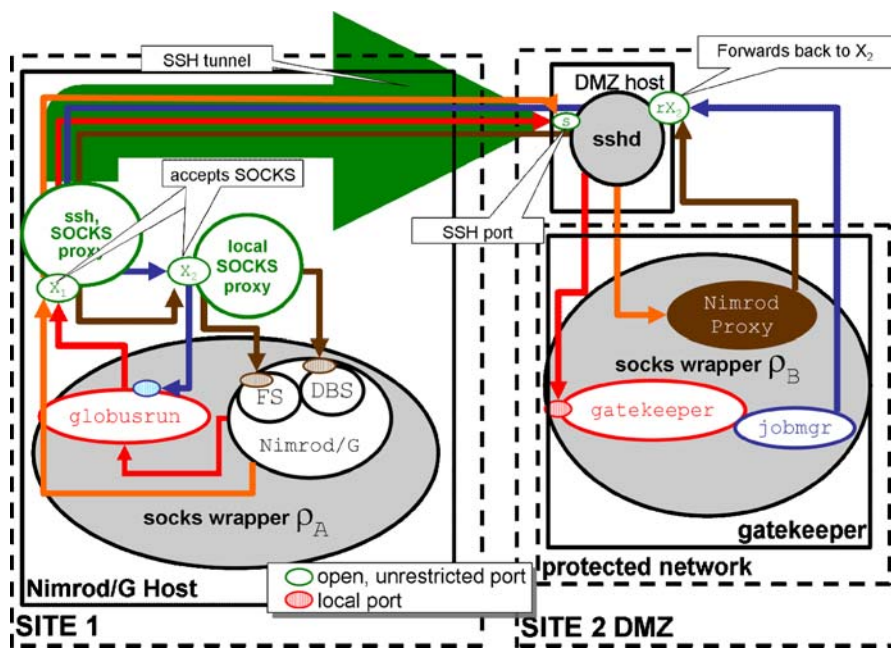


Fig. 7 Nimrod/G server and agent components are connected via REMUS. The underlying connections are as in Fig. 5, except that Nimrod/G components are also wrapped around SOCKS

firewall, in the sense that Grid middleware and applications can be viewed as applications on top of the standard networking infrastructure.

In examining the security aspects of Grid applications using REMUS, we must consider three levels:

1. The Grid security level pertains to Grid-specific mechanisms and protocols that enforce authentication and authorization. Globus Grids, for example, validate all transactions using X.509 credentials that must be authorized by the VO's certificate authority. For example, there must be valid user, host, and proxy certificates.
2. The REMUS security level pertains to REMUS-specific mechanisms and protocols that we enforce in two ways. First, REMUS cannot be used arbitrarily to get into a secure network. Creating a rerouted path through a tunnel requires a valid, authorized and authenticated account in the remote site. In our SOCKS + SSH implementation, the tunnel is erected through an SSH session, itself requiring a valid account that must accept dynamic port forwarding. This implementation relies on strong authentication techniques used by SSH, such as the use of public key rather than symmetric key encryption. Port forwarding can be switched off for users who are allowed to remotely log in but are not authorized to perform port forwarding. Furthermore, while our current implementation uses SOCKS v4, future implementations will use SOCKS v5, itself requiring certificate-based authentication and authorization. The second way by which to enforce security in REMUS is by managing a "rerouting table". Like a normal IP routing table, the REMUS rerouting table controls the path to be taken by connections from X to Y. Unlike normal IP routing tables, however, apart from source and destination IP addresses, rerouting tables must also consider source and destination ports. The level of control is similar to that which is used for typical packet filtering, i.e., stateless IP firewalls.
3. Finally, security is still ultimately controlled at the network infrastructure level. We may consider REMUS to be a middleware that is inserted between the Grid middleware and the underlying network infrastructure. This typically involves TCP/IP networking, and REMUS will not be able to enter a network if its firewall does not authorize entry. For example, using the SOCKS + SSH implementation, an SSH session must first be created in order to create the tunnel. This requires access to the SSH service on port 22, as well as an explicitly granted user account. REMUS is not intended to break firewalls by entering networks without authorization. It must be configured to use only authorized means, such as through a valid SSH session for the user.

There is no doubt that REMUS is lowering security to some extent. It may be subjected to abuse, perhaps by unauthorized users. However, the alternatives can be worse. As mentioned previously, Globus toolkit developers recommend opening ten ephemeral ports on the client site and 20 on the server site—for *each* concurrent user. We believe that REMUS presents a better alternative.

8 Performance Tests

While rerouted networking through REMUS is expected to be slower than direct networking, we decided to test our throughput anyway, to see how far behind REMUS performance might be. Note that these tests were conducted in an environment where SSH is the only channel open between the two networks. Therefore, while our performance might normally be considered poor, our results should be seen in the proper perspective. We are not claiming that REMUS improves throughput, but we do show that REMUS succeeds in providing connectivity where there was previously none possible (without removing some firewall restrictions).

8.1 Test Environment

Note that the details below are indeed for a specific implementation of REMUS. We noted earlier in the philosophy of REMUS that it is meant to be capable of different modes of connectivity.

These tests are for a specific mode that is suitable for typical secure environments:

- SSH is the only accessible service across, so we use SSH to create the tunnel.
- In order to multiplex several connections across the tunnel, we use a SOCKS proxy.

Therefore, when we refer to test details below in terms of REMUS, the details are actually specific to the SOCKS over SSH implementation—as far as the transport is concerned. The rerouting layer of REMUS, which is implemented.

The two hosts are about 11 km apart, connected via a combination of copper and fiber. The average round-trip time is 0.43 ms. Both hosts use network interfaces that operate in full-duplex at a theoretical throughput of 1,000 Mbps (Gigabit Ethernet). They are about two routers apart from each other, the average bandwidth between them is 94.4 Mbps, the average round-trip time is at 0.475 ms, and the average jitter at 0.1685 ms. On the other hand, our graphs are not meant to prove any sort of performance improvement. We are aware of how tunneling will introduce performance degradation, due to the aggregate delays introduced by encapsulating TCP (the stream we are tunneling) within TCP (the tunnel stream). The graphs are provided to compare stream-oriented (as against burst-oriented traffic) throughput using REMUS rerouting and tunneling against the theoretical ideal, in which connections are made directly. The point is simply that REMUS does introduce throughput degradation, but only by as little as 25% from the best case we've encountered. It must also be pointed out that the tests were conducted on a live network in the presence of typical heterogeneous traffic (as can be found in a multi-campus university).

8.2 Results

While the two hosts are within the same organizational domain, they are connected over a wide-area network. Figure 8 and Table 1 show how data transfers will slow down due to the rerouting that REMUS uses to divert connections across the firewall. Measurements are in Megabytes per second, computed from the size of the file transferred and the best time taken from several transfers. We performed tests to transfer files with sizes of 50, 100, 150, 200 and 250 Mb, all containing binary data. This is significant when considering that it may reduce the effectiveness of compression in the transport.

1. *Top Graph*, *Mode1*: *Direct*. The topmost graph represents direct GridFTP transfers. Four parallel GridFTP streams were used. This mode of transfer, with the firewall temporarily relaxed for purposes of this study, presents the best performance, with almost 5 Mb transferred per second.
2. *Middle Graph*, *Mode 2*: *REMUS Rerouted and Tunneled*. As with the previous graph, there are four parallel GridFTP streams for each file transfer at the application layer. At the transport layer, only one TCP stream is actually used across the network by the tunnel. SSH internally uses individual "channels" for fcs presents our best case so far, nevertheless suffering a 25% reduction in network performance compared to the best case (direct). Due to the multiplexing, plus the latency introduced by intercepting connections, rerouting them, and encrypting the streams, transfer rates shrink to about 3.8 Mbps.
3. *Lowest Graph*, *Mode 3*: *REMUS Rerouted Only*. As with the first case of direct transfers, there are four parallel GridFTP streams at the application layer. There are also four actual TCP streams at the transport layer. Here performance is at its worst among the three, peaking at only 2.8 Mb transferred per second. That Mode 2 performs better than Mode 3 can be attributed to OpenSSH. Both modes use SOCKS, equally suffering from the same overhead of SOCKS encapsulation, i.e., headers, SOCKS request/response dynamics. On the other hand, whereas Mode 3 uses a SOCKS proxy on the remote site, Mode 2 situates the SOCKS proxy locally. In effect, the SOCKS encapsulation overhead bytes are not transported in Mode 2. This can be gleaned from the source code of OpenSSH (we examined 4.7p1). Accordingly, the application

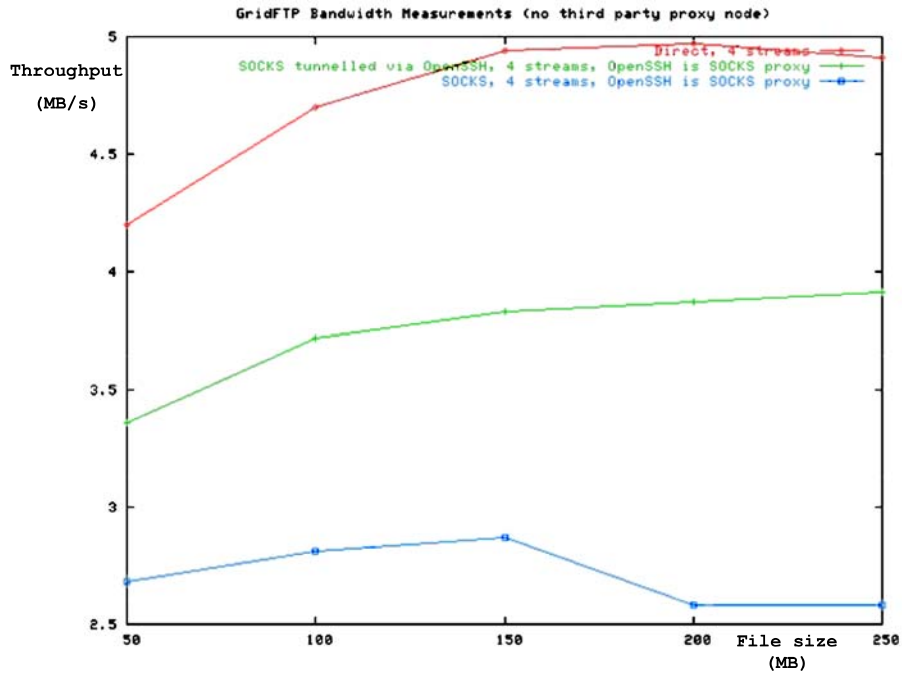


Fig. 8 Throughput graphs for Direct, REMUS (SOCKS + SSH tunneling) and pure SOCKS, in that order from the *top* to the *bottom* graphs, in terms of megabytes per second of data transferred using GridFTP

data encapsulation efficiency of SOCKS is at about 84% [17]. However, for each of the four GridFTP streams, the actual efficiency of what the tunnel will transport in Mode 2 goes up closer to 100%, since OpenSSH does not need to send the headers down the SSH stream. Additionally, our tunnel was configured for encryption.

Using SOCKS through OpenSSH thus presents our best case so far, nevertheless suffering a 25% reduction in network performance compared to the best case (direct). There may be other ways to reduce the degradation. We have already tuned the SSH parameters in our experiments with a faster cipher, blowfish, but there may be faster cipher implementations that we haven't tried. SOCKS v5 may yield better performance at the rerouting stage compared to SOCKS v4. Other methods to optimize the connections might also minimize the degradation, and they can be the subject of further work. For example, we can use multiple tunnels. While a port bottleneck remains on port 22, which is the standard SSH port, there will be multiple SSH server threads running behind it for each tunnel stream. Such measures can offer foreseeable performance improvements up to a certain extent, but will typically continue being slower compared to direct connections. On the other hand, when secure networks are involved, direct connections may simply be impossible, as we encountered in our case studies. That REMUS would perform as well as direct connections was never our claim. Throughput degradation is expected, but unobtrusive cross-firewall communication between Grid applications becomes possible. The experience also taught us the following:

- REMUS connections can be optimized using a combination of compression and cipher configuration choices. The implementation

Table 1 Throughput figures for GridFTP tests using the SOCKS implementation of REMUS show rerouting degradation can be as low as 25% of Direct transfers

Transport	Transfer rate
Direct GridFTP	Up to 4.97 Mbps
REMUS rerouted and tunneled (local SOCKS proxy)	Up to 3.91 Mbps
REMUS rerouted only (remote SOCKS proxy)	2.8 Mbps
REMUS rerouted only (remote SOCKS proxy, alternative implementation)	1.8 Mbps

also has an effect. We use Dante to "socksify" connections in both modes 2 and 3 as described above, but using a remote Dante SOCKS proxy brought performance down to less than 1 Mbps. The Mode 3 figures in the previous table come from the use of OpenSSH as the remote SOCKS proxy, with slightly better performance at about 1.8 Mbps (last entry in Table 1). Other implementations of SOCKS may provide even better results, but finding or building the best, with the most optimized configuration, will require some experimentation.

- It is best to situate the SOCKS proxy at the host from which the traffic originates, rather than at the destination host. As described earlier, we tried using a SOCKS proxy at the remote host in Mode 3, connected to directly (port 1080). We also tried a Dante SOCKS proxy accessible via a straight port-forwarding tunnel from the source host, availing therefore of the compression in the tunnel. Either way, there was throughput degradation, as the bottleneck appears to be the SOCKS dynamics and encapsulation between the SOCKS client and the SOCKS proxy. We yielded better results in Mode 2, where the SOCKS proxy was in the same host as the component sending data, and the encapsulation overhead can be excluded from the actual transport between networks.

9 Related Work

Many have studied the problem caused by firewalls that hinder successful Grid operations [4–6, 12, 13, 19, 21, 22, 26], particularly for Globus-based Grids. The Globus toolkit firewall requirements document [28] defines the access settings necessary to facilitate successful Globus transactions. A large range of ports must be accessible in both directions, whereas security policies tend to generally restrict the number of open ports. In many cases, it is simpler to use tunnels through those few open ports, if possible, rather than to open several ports. An early and common example of using tunneling is through SSH port forwarding [12]. SSH is commonly accepted through firewalls to some hosts in the secure network because SSH is itself hardened against unauthorized access. Luckily, SSH provides port forwarding as a means to route several channels of traffic through a single connection. However, tunneling Globus transactions this way is somewhat unwieldy. It is necessary, for example, to explicitly include port numbers in Globus toolkit command line arguments. It is also necessary to sacrifice callbacks from Globus components, which won't work if direct connections are blocked by firewalls protecting those remote sites. This can be mitigated to a certain degree, e.g., using batch submissions, but giving up callback functionalities should not be necessary, as we illustrate with REMUS.

Other customizations of Globus, such as the *Nexus Proxy* that can sit outside the firewall [26], are undesirable because of those customizations. Likewise, users may also resort to reprogramming Globus through hard-coded settings, such as one case involving GridFTP running on a private network that was subject to network address translation (NAT) to remote hosts on the Internet [27]. One Grid programming environment we surveyed, *NetIbis* [4], uses a specialized network infrastructure underneath that uses *simultaneous initiation* or *TCP splicing*, a feature included in the TCP standard in which both parties establish connections with each other. Such reprogramming and customizations should not be necessary. Legacy or pre-existing applications and Grid implementations need not be reprogrammed. *Generic connection brokering* (GCB) uses a broker, accessible to both parties, which will make the decision as to the direction for the connection between parties [23]. This allows a connection to proceed between two components in either direction, even if firewall restrictions would normally block connections in one direction or the other. The use of the GCB that is accessible to outgoing connections from connecting hosts is similar to how REMUS makes use of rerouters in proxy nodes. However, compared to the SOCKS wrapper implementation of REMUS, GCB is not as unobtrusive, as it does not use the notion of wrappers to completely hide the presence of a third party relay. *XRAY* [22] is another facility that uses the *GCB broker* in order to facilitate communications between client and server applications.

XRAY uses agents on behalf of clients and servers to minimize customizations to applications and middleboxes. *Owner tests* determine benign and legitimate connections while also using the defensive features of available middleboxes, e.g., firewalls. XRAY manages rules to allow or deny connections to or from sockets of client and server applications after successful authentication and authorization tests. To facilitate connections to servers, the agent for a successfully registered server application obtains an official IP address through a GCB *inagent* which leases the required sockets on the GCB broker. Client applications are informed of the officially bound addresses so that connections are made to the GCB broker instead. Client and server certificates are also required for successful connections between client and server. REMUS is architecturally compatible with XRAY, with similar recourse to agents and relaying via mutually accessible proxies. On the other hand, REMUS ignores middleboxes and our network model assumes that middleboxes are relevant only at the fabric level.

There is a particular class of solutions to the firewall traversal problem which views the firewall and other middleboxes as a controllable resource, sometimes referred to as a *dynamic firewall* [6]. Intermediary devices that sit between communicating components in a network are called middleboxes, and the notion of *middlebox communications* [19] standardizes them as controllable resources. There is on-going effort pushing for an appropriate IETF standard for "midcom". However, there are also non-standard efforts along the lines of controllable firewalls. Cooperative on-demand opening, developed by the people behind the XRAY project, controls firewalls directly by altering the in-kernel firewall configuration in order to admit connections if verified to be legitimate [21]. Tests are conducted to authenticate and authorize applications before lifting the restrictions as required. Owner tests and application registration is essential given that the firewall itself is being controlled through agents. The dynamic firewall (*Dyna-Fire*) project within the Advanced Computational Data Center Grid (ACDC-Grid) also makes the firewall reconfigurable at [13]. Connections are only established after a successful "knocking" phase when clients authenticate themselves through an authorized secret "knock". The philosophy is good but is not without disadvantages. A number of ports on the firewall must be reserved for port knocking communication, and cost of monitoring knocks is non-trivial. There is also concern about the port knocking implementation and certain security exploits, i.e., message replay and man-in-the-middle attacks [6]. However, Dyna-Fire will only trust a knock from a trusted IP address, but this makes client authentication dependent on IP addressing, rather than encrypted and/or signed credentials.

Other forms of tunneling mechanisms have been built but for other purposes, such as glogin [20], which erects a tunnel through a job submitted as a normal Globus job. This is intended to provide interactivity to users, giving them remote login access to Grid nodes. It will not work if the firewall blocks connections between the user hosts and the glogin job on the compute node, or if jobs are constrained by maximum runtimes.

An impressive connectivity infrastructure that we may compare against is WOW [10]. Real world hosts execute virtual machines which behave as virtual Grid workstations. WOW provides the virtual addressing and location facilities in order to form a virtual network of Grid resources. These are able to interconnect across firewalls through a peer-to-peer (P2P) overlay network sitting outside secure domains. However, as with any solution, the firewalls must allow at least outgoing connectivity to the P2P network, which facilitates tunnels in order to provide Grid networking across secure domains of different institutions. This would not have worked with the Royal Melbourne Institute of Technology (RMIT) student lab scenario, which we briefly describe below in the conclusion section. The P2P overlay network therefore performs the functions of a proxy facility. WOW provides several important facilities, including a robustness that allows virtual workstations to migrate from one real world network to other real world networks. WOW is also mostly unobtrusive, since virtual workstations exist in a totally sandboxed guest environment on the host that executes it. However, these features come with a price. The virtual machine approach is costly, since virtual workstations are executing as a complete guest operating system. WOW

virtual workstations are complete Linux systems. They only require a minimal Linux installation, which excludes desktop utilities but includes a full kernel, a basic set of device drivers, and Grid middleware. Disk storage requirements are small, but memory requirements are as would be required of a Grid resource. More importantly, each virtual workstation executes as many heavyweight and lightweight processes as one would expect in a standalone host running Linux. This has considerable CPU cycle implications for the machine hosting the virtual workstation. In contrast, REMUS uses the middleware approach. On a given Grid resource, REMUS introduces a middleware tier underneath Grid middleware such as REMUS. Both memory and processor impact are minimal. As far as the viability of firewall traversal is concerned, REMUS and WOW are comparable. They both require a reasonable firewall situation, in which connections are possible with proxies (in the case of REMUS) or P2P overlay networks (in the case of WOW). In the data transfer measurements of WOW, throughput was between 1.36 and 1.83 Mbps, using SSH (secure) copy. REMUS throughput, via GridFTP transfers using SSH tunnels, reached more than 3.75 Mbps. However, we cannot accurately compare throughput since data concerning network distances between hosts and background (unrelated) loads across networks between hosts are not available. While we can measure the degradation of throughput between direct and REMUS-facilitated data transfers, no such measurement was given for the hosts involved in the WOW data transfer tests.

REMUS prefers to leave firewalls alone. Middlebox devices are viewed as fabric resources which need not be visible at the Grid connectivity level. Implementing REMUS rerouting via SOCKS and SSH provides an adequate degree of security since SSH connections require legitimate user accounts that are authenticated with asymmetric key cryptography. REMUS SOCKS proxies run at the user level and are configured individually for each user. In essence, connections are legitimized by the fact that the users establishing those connections are legitimate users with accounts and credentials authenticated at the system level, i.e., via SSH logon. Note that it is also possible to use X.509 certificates for SSH logons using implementations of SSH that support GSSAPI [14]. We must also note that controllable middleboxes within a network are not yet typical, apart from direct super-user commands to control the firewall or router. Middlebox recognition of Grid-based credentials is largely unsupported at this time, except in cases where firewall and routing mechanisms are incorporated into firewall traversal systems, e.g., [13, 21, 22]. Firewall reprogramming at runtime must be approached carefully, given the potential risks involved, even for short-duration tampering. It also presents a blurring of fabric and connectivity components, where middleboxes like firewalls and NAT gateways become visible at the Grid level, i.e., they become Grid resources. Network policies will require substantial rethinking in order to make such blurring satisfactory across the entire organization. There are also disadvantages when relying on proxy X.509 certificates for a temporary lifting of firewall restrictions. Some proxy certificates may be valid for weeks, such as those used in long-running scientific computations. While REMUS does not disagree with the notion of firewall controls at the Grid-level, we are working with what are currently typical situations.

10 Conclusion

Figure 9 illustrates what we have achieved with REMUS. As a "lower middleware", REMUS implements a virtual connectivity layer that enables Grid communications on top of the Fabric Layer's TCP/IP stack, without breaking the restrictions imposed by firewalls. We implemented a rerouting architecture that uses wrappers to facilitate connections across firewall restrictions, and tested it across a VO setting with a firewall between two participating institutions' networks. One institution hosts Nimrod/G and accepts experiments, while the other institution hosts Globus computational resources. Our latest implementation of REMUS provided basic connectivity using SSH to tunnel connections through, with SSH being the open channel authorized by both institutions. In order to handle multi-port communications, we used SOCKS proxy connections tunneled through the SSH session. To provide components with

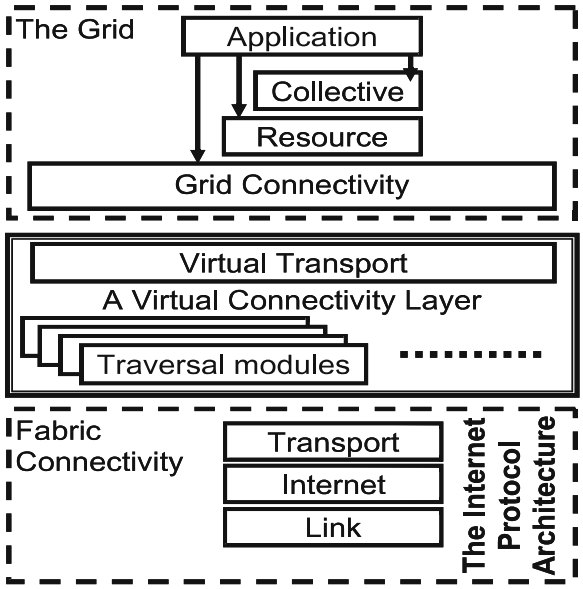


Fig. 9 REMUS is an implementation of a virtual connectivity layer that supports general-purpose Grid communications across firewalls

SOCKS capabilities quickly, we used SOCKS wrappers that automatically reroute connections via SOCKS across the SSH tunnel. Importantly, *there was no need to modify any piece of code for both Globus and Nimrod/G*, nor was there a need to modify the firewall settings. The libraries used for SOCKS communication were dynamically linked and simply preloaded using environment variable settings. This version of REMUS also demonstrates how GSI authentication succeeds despite the actual port dynamics involved in SOCKS-wrapping and tunneling.

To date, we have had two Nimrod/G and Globus installations outside of Monash University that were assisted by REMUS. A normal student lab at RMIT University in Melbourne, Australia, was configured as a Globus resource running Condor for job management. Being inaccessible from outside, we created two SSH + SSH tunnels between Monash and RMIT using a DMZ host as a proxy node, as shown in Fig. 7. We used library preloading to place all Nimrod and Globus processes within a SOCKS-enabled environment. The second case involved the South Australian Partnership for Advanced Computing, which also ran a cluster within a protected network. Globus connections were accepted through a gateway node that was outside of these restrictions. However, Nimrod agents executing on the cluster nodes were unable to connect back to the Nimrod servers outside. They implemented REMUS with pure SOCKS (no SSH) so that the Nimrod agents are rerouted to access Nimrod servers outside.

In the near future, we will further investigate the performance implications of REMUS. We do not have access to performance data from the two installations just mentioned. They have discontinued use of these facilities for reasons unrelated to Nimrod/G, Globus and REMUS. At present we only have access to REMUS installed across two campuses of Monash University. We are looking to install REMUS on other live sites, where we can test the efficacy of parallelizing connections through multiple proxies and multiple tunnels. We will also explore and implement other means of traversing firewalls, since different networks configure different sets of open ports. REMUS is envisioned to provide a multi-pronged approach, with multiple rerouting capabilities to suit different scenarios. We must also investigate the applicability of REMUS in facilitating UDP connections, something which is likely, using SOCKS v5, but has yet to be attempted in REMUS. Finally, we will improve the internal architecture of REMUS rerouters to make it easier for them to adapt to whatever deployment scenario they encounter, through semi-automation features, policy translation [18], or through easy configuration directives.

Acknowledgements The Australian Partnership for Advanced Computing supported this work. Prototype experiments with REMUS were made possible with the kind assistance of the South Australian Partnership for Advanced Computing and Dr. Savitri Bevinakoppa of Royal Melbourne Institute of Technology University, Melbourne.

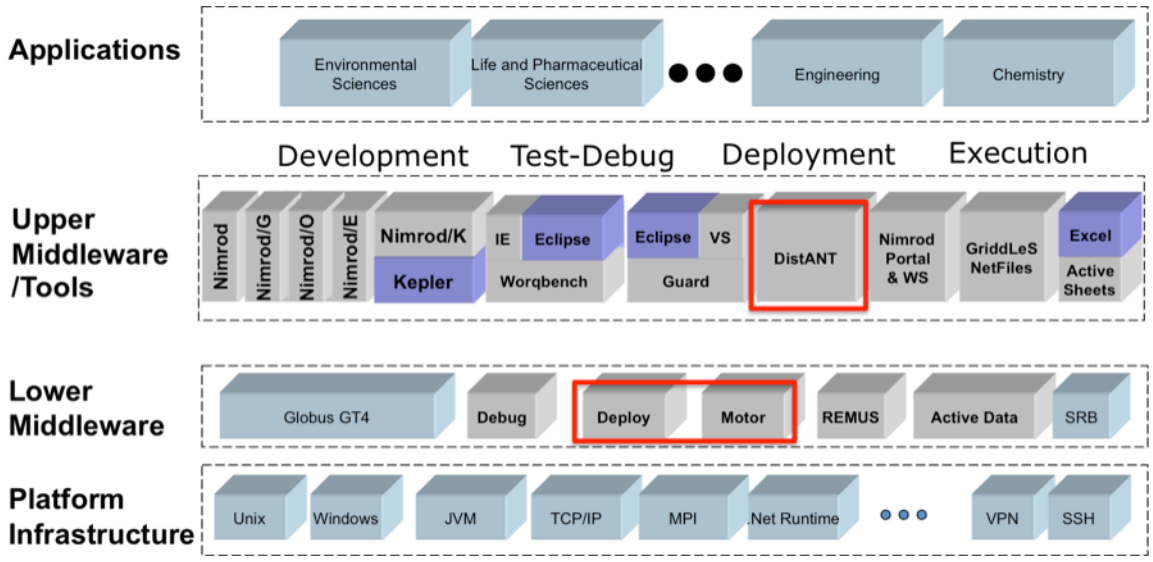
References

1. Abramson, D., Sosic, R., Giddy, J., Hall, B.: Nimrod: a tool for performing parametised simulations using distributed workstations. In: Proceedings of the 4th

IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995
2. Buyya, R., Abramson, D., Giddy, J.: Nimrod/G: an architecture of a resource management and scheduling system in a global computational Grid. In: Proceedings of HPC Asia 2000, Beijing, China, 14–17 May 2000
3. Comer, D., Stevens, D.: Internetworking with TCP/IP volume 3: client-server programming and applications. Prentice-Hall, Upper Saddle River (2001)
4. Inferno Nettverk A/S: Dante—a free SOCKS implementation (Online). http://www.inet.no/dante/. Accessed 2005
5. Denis, A., Aumage, O., Hofman, R., Verstoep, K., Kielmann, T., Bal, H.E.: Wide-area communication for grids: an integrated solution to connectivity, performance and security problems. In: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13 '04), pp. 97–106 (2004)
6. Dynamic Firewalls Configuration, Ver. 1.0, D-Grid Integrationsprojekt (DGI), Technical Report, (Online). https://www.d-grid.de/fileadmin/user_upload/documents/DGI-FG3–5/FG3–5_Dynamic_Firewalls.pdf. Accessed Feb. 2006
7. Foster, I., Kesselman, C., Tsudik, G., Tuecke, S.: A security architecture for computational Grids. In: Proceedings of the 5th ACM Conference on Computer and Communications Security Conference, pp. 83–92 (1998)
8. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the Grid: enabling scalable virtual organizations. Int. J. Supercomput. Appl. **15**(3), 200–222 (2001)
9. Foster, I., Kesselman, C.: Globus: a metacomputing infrastructure toolkit. Int. J. Supercomput. Appl. High Perform. Comput. **11**(2), 115–128 (1997), Summer
10. Ganguly, A., Agrawal, A., Boykin, P.O., Figueiredo, R.J.: WOW: self-organizing wide area overlay networks of virtual workstations. J. Grid Comput. **5**(2), 151–172 (2007)
11. The Globus Alliance: Globus Toolkit. http://www.globus.org. Accessed 2008
12. Graupner, S., Reimann, C.: Globus grid and firewalls: issues and solutions in a utility data center environment, Technical report HPL-2002-278. Hewlett-Packard, Palo Alto, 2 October 2002
13. Green, M.L., Gallo, S.M., Miller, R.: Grid-enabled virtual organization based dynamic firewall. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID'04), pp. 208–216 (2004)
14. Hutzelman, J., Salowey, J., Galbraith, J., Welch, V.: Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol, RFC document, IETF (Internet Engineering Task Force). http://www.ietf.org/rfc/rfc4462.txt. Accessed May 2006
15. Lee, Y.: SOCKS: a protocol for TCP proxy across firewalls, NEC Systems Laboratory (Online). http://archive.socks.permeo.com/protocol/socks4.protocol. Accessed 2005
16. Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., Jones, L.: SOCKS protocol version 5. IETF RFC 1928. http://tools.ietf.org/html/rfc1928. Accessed March 1996
17. Mills, W.N. III, Krueger, L., Chiu, W., Halim, N., Hellerstein, J.L., Squillante, M.S.: Metrics for performance tuning of web-based applications. Computer Measurement Group, Turnersville (2000)
18. Neisse, R., Bosquiroli Almeida, M.J., Granville, L.Z., Rockenbach Tarouco, L.M.: Policies translation for integrated management of grids and networks. In: Liebrock, L.M. (ed.) Proceedings of the 2005 ACM Symposium on Applied Computing, pp. 724–729, March 13–17, Santa Fe, New Mexico. SAC '05. ACM, New York (2005)
19. Quittek, J., Stiemerling, M., Srisuresh, P.: Definitions of Managed Objects for Middlebox Communication, Internet-Draft, IETF (Internet Engineering Task Force), October 2006
20. Rosmanith, H., Kranzlmuller, D.: Glogin - a multifunctional, interactive tunnel into the grid. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID'04), pp. 266–272 (2004)
21. Son, S., Allcock, B., Livny, M.: CODO: firewall traversal by cooperative on-demand opening. In: Proceedings of the 14th IEEE Symposium on High Performance Distributed Computing, July 2005
22. Son, S., Farrellee, M., Livny, M.: a generic proxy mechanism for secure middlebox traversal. In: Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2005). Boston, Massachusetts, USA, 27–30 September 2005
23. Son, S., Livny, M.: Recovering internet symmetry in distributed computing. In: Proc. 3rd IEEE/ACM International Symposium Cluster Computing and the Grid (CCGrid 2003), pp. 542–549, 12–15 May 2003
24. Sudholt, W., Baldridge, K., Abramson, D., Enticott, C., Garic, S.: Application of Grid computing to parameter sweeps and optimizations in molecular modelling. In: Zomaya, A. (ed.) Parallel Computing for Bioinformatics, pp. 27–35. Wiley, Hoboken (2006). Also appeared in The International Conference on Computational Sciences (ICCS04), Krakow Poland, 6–9 June 2004
25. Tan, J., Abramson, D., Enticott, C.: Bridging organizational network boundaries on the Grid. In: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid 2005). Seattle, Washington, 13–14 November 2005
26. Tanaka, Y., Hirano, M., Nakada, H., Sekiguchi, S.: Performance evaluation of a firewall-compliant Globus-based wide-area cluster system. In: Proceedings of the IEEE International Symposium on High-Performance Distributed Computing 2000 (HPDC 2000), pp. 121–128, 1–4 August 2000
27. Thorpe, S.: Online, an email sent to the globus-discuss mailing list about running a GridFTP server behind a NAT firewall. http://www-unix.globus.org/mail_archive/discuss/2005/01/msg00216.html. Accessed January 2005

28. Welch, V.: Globus Toolkit Firewall Requirements, version 9. The Globus Alliance, (Online). http://www.globus.org/toolkit/security/firewalls/Globus-Firewall-Requirements-9.pdf. Accessed 31 October 2006
29. Welch, V., Foster, I., Kesselman, C., Mulmo, O., Pearlman, L., Tuecke, S., Gawor, J., Meder, S., Siebenlist, F.: X.509 proxy certificates for dynamic delegation. In: Proceedings of the 3rd Annual PKI R&D Workshop (2004)
30. Ylonen, T.: SSH—Secure login connections over the internet. In: Proc. 6th USENIX Security Symposium, San Jose, CA, pp. 37–42, July 1996

Chapter 14: Grid Deployment and Virtualization



Much of the Grid middleware addresses issues of how to invoke services and access data on remote platforms. A much-ignored issue is how to deploy applications in the first place. We experienced this problem many times with Nimrod, and developed a deployment strategy based on resource virtualization. (57) discusses a general framework for doing this, and (58) addresses support for parallel programs on the .NET runtime.

(1) Goscinski, W. and Abramson, D. "An Infrastructure for the Deployment of e-Science Applications", in "High Performance Computing (HPC) and Grids in Action", Volume 16 Advances in Parallel Computing, Editor: L. Grandinetti, March 2008, approx. 540 pp., hardcover, ISBN: 978-1-58603-839-7, pp 131-148. 14-1 .. 14-18

(2) Goscinski, W. and Abramson, D. "Parallel programming on a high-performance application-runtime", Concurrency and Computation: Practice and Experience. June 2008. doi:10.1002/cpe.1325. 14-19 .. 14-55

Contributions

Goscinski was a PhD student under my supervision.

High Performance Computing and Grids in Action
L. Grandinetti (Ed.)
IOS Press, 2008

© 2008 The authors and IOS Press. All rights reserved.

An Infrastructure for the Deployment of e-Science Applications

Wojtek James GOSCINSKI and David ABRAMSON
{wojtek.goscinski, david.abramson}@infotech.monash.edu.au
Faculty of Information Technology, Monash University

Abstract. Recent developments in grid middleware and infrastructure have made it possible for a new generation of scientists, e-Scientists, to envisage and design large-scale computational experiments. However, while scheduling and execution of these experiments has become common, developing, deploying and maintaining application software across a large distributed grid remains a difficult and time consuming task. Without simple application deployment, the potential of grids cannot be realized by grid users. In response, this paper presents the motivation, design, development and demonstration of a framework for grid application deployment. Using this framework, e-Scientists can develop platform-independent parallel applications, characterise and identify suitable computational resources and deploy applications easily.

Keywords. Application deployment, application runtime, grid middleware

Introduction

A computational grid is a High Performance Computing (HPC) infrastructure that enables the development and execution of a new generation of large-scale, multi-domain and multi-institutional e-Science experiments [1]. This new environment, though incredibly promising, poses particular challenges to end-users. One of these challenges is the deployment of software across a computational grid [2, 3].

In order to perform a grid experiment, an e-Scientist first builds an *application testbed* by installing necessary application software across selected resources. While the execution and orchestration of large-scale experiments is becoming more common, users are not provided with a mechanism to install required software across grid resources. Large-scale, dynamic, user-oriented deployment is not supported by current grid middleware. This is a significant problem because the potential scale and highly heterogeneous nature of a grid makes effective manual deployment difficult or even impossible.

In order to most effectively utilize a large-scale grid environment, we believe users require tools and services that allow them to define and realise the largest and most relevant application-testbed. In response, this paper presents the motivation, design, development and demonstration of a framework for grid application deployment. This framework combines three layers of middleware: a user-oriented environment that enables grid-scale application deployment; a platform and data transport independent application-runtime which simplifies deployment over heterogeneous resources; and a deployment tool, which automatically installs an executable and the required library modules.

Using our framework, users can develop platform-independent parallel applications. They deploy those applications using a deployment tool which can characterise resources to form the largest and most relevant testbed. Deployed applications can then be addressed and executed using a deployment reference. As a consequence, users are presented with an overall environment which simplifies the steps required to realize a large-scale grid experiment.

1. Motivation

Consider the following scenario:

An e-Scientist with a specialization in biotechnology, and reasonable programming experience, develops a computational model to represent a biological system in the human body. Her experiment is computationally intensive, so strong performance is a high priority. However, considering her specialty in biotechnology rather than computing, she considers simple software development to also be of high importance.

She has access to a large set of resources through an inter-university grid but needs to select specific resources which provide the necessary computational, network and storage services. From her point of view, resources are uncharacterised; she knows little about their individual components such as operating systems, architectures or libraries. Her deployment and execution strategy is dynamic and opportunistic; she does not know which resources will be available at the time of execution and she wants to use the most appropriate resources of those available.

An analysis of first results shows a small bug in her code. She fixes the bug, recompiles and then redeploys over the entire set of resources. Analysing the results, she continues improving, extending the program and redeploying, to create a consistently better model.

This simple scenario is relatively difficult to realize with present-day grid infrastructure for a number of reasons:

1. Large-scale, dynamic, user-oriented deployment is not supported in current grid middleware [2, 3]. Users must resort to highly user-intensive command line tools to copy, compile, configure and test software. Deployment is an overlooked feature of grid middleware, but is important for a number of reasons.

   First, the potential scale of application testbeds means it is not feasible to install software on one resource at a time. Second, grid access does not guarantee that a software package can be successfully installed. For example, a grid certificate does not usually allow a user to login to a resource. Furthermore, different resources provide different levels of access, according to the administration policy. For example, while some resources might provide a user with relatively free reign, others might implement strict controls and limitations. Third, users might have access to a grid, but know little about individual resources making it difficult to discover or select appropriate resources.
2. Users are restricted by the vast technical and organizational heterogeneity of the grid. Developing and deploying high performance programs, over a range of heterogeneous resources is a difficult and time consuming process [2-5].

Technical heterogeneity complicates application development and deployment. Standardization of high performance communication libraries, such as MPI, simplifies development for different processors and interconnects. However, programmers still face an assortment of non-MPI porting and installation issues, such as platform specific system calls, recompilation and library problems. Likewise, installation is difficult due to different build and executable files, library dependencies, file systems, build procedures and installation procedures. Furthermore, software usually requires a different installation or build sequence from one resource to another. This is because different resources will support different versions of tools, libraries and compilers. Managing software dependencies in this environment can be a major undertaking.

Further, grids are also organisationally heterogeneous, that is, resources have different administration, operating procedures and access policies. Identical hardware running identical middleware will invariably be configured differently by different system administrators, let alone different institutions. This complicates deployment due to different access, permissions, accounts and installation procedures.

3. Parallel programs are most commonly developed natively; they are written and compiled directly for a specific operating system and instruction set. This approach provides the highest level of performance and control. However, developing and deploying native software is challenging for two reasons:

   First, native code requires porting for different platforms. This is a major undertaking considering the potential complexity of e-Science software and the range of platforms which might constitute a grid. Furthermore, native code needs to be recompiled for each platform, which itself is a significant effort. This limits users when making deployment and execution decisions. Second, native binaries alone provide no mechanisms for defining library dependencies. Users need to know an application's dependencies and manually install them.

   This is in contrast to the current trend in software engineering, which has moved toward an application-runtime architecture, such as that provided by Java [6] or the Common Language Infrastructure [7] and its commercial implementation, Microsoft .Net [8].

   Typically, native code is faster in comparison to code managed by a runtime environment. On the other hand, runtime environments provide a range of services which have proved successful in fostering software development in the general areas of business and desktop computing. These include: compile-once-run-anywhere development, a common type system, platform-independent self-describing assemblies and a guaranteed set of libraries.

   Most significantly, runtime environments provide an execution guarantee. In essence, a runtime specification defines a contract between user code and the runtime. As a result, code is guaranteed to execute on a correctly implemented runtime, so long as both parties adhere to the contract. This allows developers to focus on functionality, rather than porting to various platforms. It is particularly significant for an e-Scientist installing software over a range of heterogeneous resources.

   Unfortunately, existing runtimes alone do not provide the necessary functionality for building HPC applications, particularly message passing and shared memory. We believe this is a significant factor which has prevented e-Scientists from using runtime environments for parallel processing.

2. Infrastructure for the Deployment of e-Science Applications

In response to the deficiencies introduced in the preceding section we present the design and implementation of an Infrastructure for the Deployment of e-Science Applications (IDEA). This framework consists of three middleware layers (Figure 1):

1. Distributed Ant (DistAnt) [2, 3], a user-oriented service-level infrastructure for grid-scale software deployment over heterogeneous uncharacterised resources. It allows developers to define and realize a testbed and consists of two major components: the DistAnt Service which provides a common deployment service (Section 2.3); and the DistAnt Workflow which is a tool for describing and enacting deployment steps (Section 2.4).
2. Motor [4], an application-runtime which provides a common environment for software development, deployment and execution. It supports a runtime-internal MPI communication library for parallel processing.
3. The Automatic Deployment Tool (ADT), a tool that automatically deploys a Motor executable and its library dependencies.

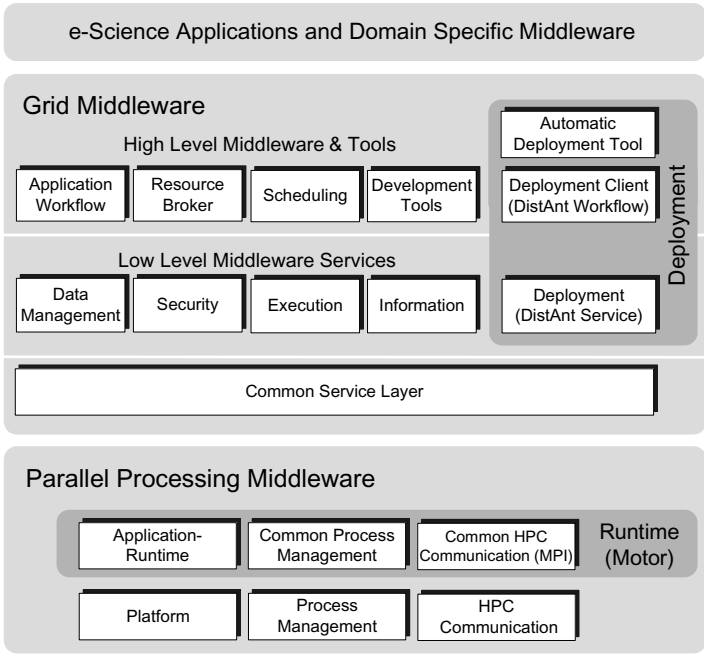


Figure 1. The multilayer IDEA framework, showing its relationship within existing grid and HPC middleware layers. Components defined by the IDEA framework are highlighted in dark grey.

The following sections describe important features and advantages of this architecture and its components. Section 3 describes the use of the whole IDEA framework in developing and deploying parallel applications.

2.1. User-Oriented Application Deployment

Application deployment should be simple; otherwise, the potential of grids will not be realized. The user-oriented nature of the grid requires that users have the ability and authority to deploy software on resources to which they have received access, but do not necessarily control. To support simple application deployment, our architecture is *user-oriented*, which means the motivation and responsibility for a successful deployment comes from the user, with support from middleware. This model is appropriate because the grid is decentralised and multi-organisational. Relying on system administrators to install software is difficult because there is no single point of control.

Our user-oriented system, DistAnt, supports automatically characterising a grid, managing heterogeneity and deploying software, thus allowing e-Scientists to form testbeds easily. The way this is achieved is described in later sections.

2.2. Dynamic Application Testbeds

A deployment system assists users in creating large and dynamic testbeds, which is important because it provides the scheduler with the largest possible choice of targets. In addition, testbeds need to be dynamic because grids are physically dynamic. For example: new resources might become available or existing resources might be upgraded, changed, replaced or retired. In the majority of cases, users have no control over the architecture or characteristics of the grid and its resources.

Currently, testbeds are relatively static and limited in size. This is because deployment, the major task to forming a testbed, is manual and cumbersome; it is difficult to create, augment or adapt a testbed. For example, the US based TeraGrid [9] consists of a static and relative small number of high end resources. Moreover, in order to try and simplify the deployment problem, the environment on these machines is fixed and heavily controlled.

Consider the example of a new cluster being introduced into a grid. For the scheduler to take advantage of this resource, the users must discover the new cluster and integrate it into their testbed, by manually installing necessary software.

A deployment system improves the outcomes of scheduling and execution because it can support the creation of larger, more relevant and more dynamic testbeds. For example, consider the following two scenarios:

- *The execution of a parameter sweep.* To minimize execution time the application needs to be deployed over as many resources as possible. An automatic deployment system could be used to install the application over a larger set of resources than would be otherwise possible manually.
- *The execution of a scientific application which requires a distinctive hardware specification.* Resources must be identified which match the specification. However, users should not be expected to know details about every resource in a large-scale grid and manually characterising resources is time consuming. An automatic deployment system could be used to characterise and select the most appropriate resources making the testbed more relevant.

In both cases, the testbed is dynamic because deployment can be revisited. Thus, a user can automatically adjust the testbed to changes in the physical grid environment.

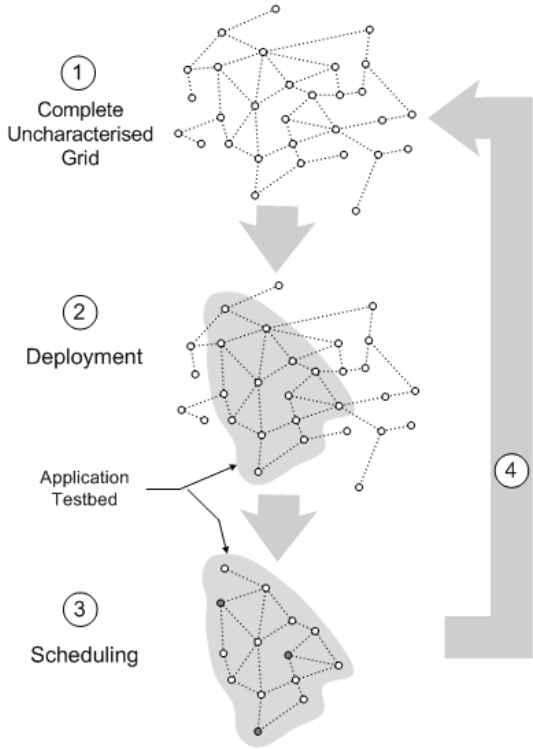


Figure 2. The interaction between automatic deployment for building dynamic testbeds and scheduling.

Our system, DistAnt, enables the following deployment, scheduling and redeployment steps (Figure 2):

1. The starting point is a completely uncharacterized grid. Therefore, the grid is large, but not necessarily relevant to the experiment because resources are unknown and do not support the required software.
2. The DistAnt system characterizes resources. It defines the testbed by selecting resources based on their physical attributes and characteristics required by the user.

   The DistAnt system realizes the testbed by deploying the necessary software over the selected resources. The testbed is relevant to the requirements of the experiment, based on the current physical state of the grid.
3. Scheduling is performed based on the runtime performance of the realized testbed, according to the appropriate scheduling strategy. Earlier steps allow the user to select resources so that scheduling is most effective. For example, the testbed for a parameter sweep should be as large as possible.
4. Revisiting deployment allows the user to update the testbed to reflect changes in the physical grid environment, software requirements, experiment requirements, or simply to upgrade software. By re-characterising the grid environment a new testbed can be formed which reflects the updated grid.

2.3. An Application Deployment Service

Much of the effort invested in developing grid middleware is devoted to defining common interfaces for resource services. The importance of this effort is demonstrated in the success of the Globus GRAM [10], which provides a common interface for execution. Similarly, deployment also requires a common service interface. Currently, to install an application, a user must log in and use supported services such as FTP, gridftp, telnet and SSH. Each resource might support different services and enforce different authentication and file system access mechanisms.

The DistAnt Service provides a common, secure access point and a standard way of performing deployment steps, such as: transporting and unpacking files, compiling and linking binaries, and enacting necessary configuration steps. Like the Globus GRAM, this service supports a user-oriented view of deployment, allowing users with resource access to install software.

The DistAnt Service also provides *deployment spaces*, which are managed local directories where users can install software. Deployment spaces can be referenced using a *deployment reference* which is a user defined handle. The combination of deployment spaces and deployment references allows users to reference an application across a set of resources regardless of the actual underlying file system location or file system structure.

The DistAnt Service can be invoked by a range of tools or clients that wish to deploy software on the hosting resource. Our design of such as tool is the DistAnt Workflow, which is described in the following section.

2.4. An Application Deployment Workflow

An application is deployed by performing a series of steps, such as file transfer, compilation, and configuration. We refer to this sequence of steps as the *deployment workflow*. Consider the following two examples:

- *A managed application with one executable binary.* In this case, the workflow needs to identify resources which support the necessary runtime and then select the most relevant resources based on the particular deployment strategy. Realizing the testbed requires copying the binary to selected resources.
- *A complex native application which is dependant on a number of libraries and can be installed from source over a range of Linux distributions.* In this case, the deployment workflow needs to identify Linux resources which support the necessary libraries and then select the most relevant resources based on the particular deployment strategy. Realizing the testbed could require a complicated series of steps on each testbed resource, such as: moving and unpacking files, performing platform specific compilation and linking, installing missing dependencies and performing a resource-specific configuration.

Performing these steps manually over a large testbed is no longer realistic. Therefore, DistAnt allows the user to describe and enact a deployment workflow. The DistAnt Workflow provides the user with a flexible mechanism for describing and enacting deployment steps, including:

- The steps for *defining* the testbed. This includes characterising and selecting grid resources. Users can write a workflow which finds those grid resources which match requirements and group them to form a testbed; *and*

- The steps for *realizing* the testbed by installing necessary software over the set of selected resources. The DistAnt Workflow provides constructs for unpacking, compiling, linking and configuring software at the remote resource. In addition, the workflow allows the user to manage resource heterogeneity. For example, it is straightforward to write a workflow which performs a different series of configuration steps, depending on the target platform.

The workflow description is built as an extension to the Ant build file system [11] and is both *procedural* and *declarative*. Procedural scripting allows the user to define deployment routines, such as the steps required to realize an installation. The declarative construct allows the user to define dependencies between routines. Each routine defines the steps for performing a particular action on the client resource *or* over a set of remote resources. Examples of specific routines might include: querying resource attributes to characterise a set of resources; forming grids into logical sets; transferring files; or enacting remote actions to perform configurations.

To support the user performing software configurations on remote resources the DistAnt Workflow provides remote routines that describe tasks such as unpacking, compiling, linking, configuring software or executing a script at the specific resource. The resource-side functionality for enacting a remote routine is provided by the DistAnt Service.

In addition, the DistAnt Workflow provides a way of managing large scale and heterogeneous grids, as discussed in the following section.

The DistAnt Workflow is a high-level middleware tool (Figure 1); it invokes a range of grid middleware services. For example, deployment functionality is provided by the DistAnt Service, file transfer is provided by a file transfer service such as GridFTP, RFT and SCP and resource information is provided by an underlying information service, such as the GT4 Index Service.

A comprehensive description of the DistAnt Workflow tool, the operations and tasks it supports, and the way a user can write a DistAnt workflow is provided in [2, 3].

2.5. Resource Heterogeneity

In order to successfully deploy software, a user needs to consider a range of resource attributes and characteristics, such as:

- The execution environment, including the operating system, architecture, configuration, compiler, binary format and data transport. Native software is often written for a specific execution environment, while code which is developed for a range of platforms must be recompiled for each platform. This is time consuming and error prone across a large testbed.
- Underlying physical resource attributes, including hardware metrics such as CPU speed, real memory size, hard disk space or distinct and specialized hardware. These attributes and metrics are important considerations when deploying software. For example, an application might require a minimum amount of real memory to execute correctly.
- HPC attributes such as cluster architecture or number of compute nodes. For example, a parallel program might need to be configured for the number of compute nodes before compilation.

- Attributes specific to the overarching experiment or workflow. For example, the file system presence of a particular data set might be a requirement of execution and therefore a prerequisite to deployment.
- Grid, organisation or resource-specific management policies. Resources will be managed differently by different organisations. For example, maximum user account disk space limits the maximum installation size.

In a typical multi-organisational grid environment, these attributes are beyond the control of the user and typical ussers will know little about individual resources, making it difficult to discover and select appropriate resources to form a relevant testbed. Furthermore, once a testbed is defined, it is likely that different resources will require a specific installation procedure. For example, software deployed on a Linux resource will require a Linux-specific compilation.

Our framework supports two approaches to developing and deploying software over heterogenous grids:

1. Native software development and deployment, with support from the DistAnt system to manage heterogeneity. This means developing, compiling and configuring software for the specific platform supported by each particular resource.
2. Managed software development and deployment. This means software is developed for Motor, our execution environment, which vastly simplifies deployment.

While the IDEA framework does include a runtime for parallel program development, we do not want to abandon users who develop native software. Developers will continue writing native parallel codes because this approach offers strong performance and highest level of control. As a result, we believe it is essential that a deployment system is not limited to managed applications, but also benefit new and legacy native applications.

It is important to note that the majority of the middleware for deploying native software remains essential for deploying managed software. Characterizing resources, selecting resources, transporting files, performing installation and configuring must be performed, regardless of whether Motor is used.

Section 2.5.1 discusses the mechanisms supported by DistAnt to manage resource heterogeneity, while Section 2.5.2 presents the alternative approach, virtualization.

2.5.1. Managing Heterogeneity using DistAnt

Using the DistAnt Workflow resource attributes are automatically discoverable and can be managed. To support this, the DistAnt Workflow defines two concepts:

- Resource queries; and
- A subgrid, which groups resources together in logical sets.

Resource Queries

The DistAnt Workflow characterises resources using a query mechanism. This is intended to be performed over a whole set of resources. For example, to discover the CPU speed of a set of resources, a user would query for CPU speed. This query would be executed for each resource in the set and the specific value would be assigned as a property to a workflow representation of each resource.

Queries are executed against a particular grid information service or other interface which can be queried for information. Currently, DistAnt supports WSRF queries. For

example, it is easy to query the GT4 Index service for resource information.

Subgrids

An effective approach for managing a large heterogeneous grid is to sub-divide it into manageable sets of homogeneous resources, which can then be treated individually. This allows the user to write routines for specific resource attributes. For example, resources which share the same platform description are likely to require a similar sequence of build steps.

The DistAnt Workflow allows the user to characterise, group and select platforms without specifically addressing individual resources. To achieve this, DistAnt implements a construct called a *subgrid*, which makes it possible to group like resources together. A subgrid is a set of zero or more resources which share specific properties. Examples of subgrids might include: the set of resources which require a particular library upgrade; or a set of resources which support the same operating system.

Defining a subgrid allows the user to write specific routines and functionality for that subgrid. For example, a user might choose to group Linux-X86 resources together in a common subgrid to perform a common Linux-X86 configuration. Significantly, a user can write a deployment workflow without addressing individual resources. Rather, the workflow can address sets of resources based on their properties. This is important because individual resources might come and go, but types of resources are likely to remain stable.

In our experience, e-Science programs are commonly installed from source code. However, a complete build from source on each individual resource is typically unnecessary. Therefore, the subgrid construct has been designed so that it is easy to build from source on one resource and redistribute binary to remaining resources. To achieve this, a subgrid can be assigned a representative resource, which means source can be compiled once on the chosen resource for each platform and then redistributed to remaining resources of that platform. Selecting a representative resource can be based on resource properties. For example, CPU speed or resource load could be used to choose the fastest resource to compile an application.

Figure 3 illustrates how the DistAnt Workflow can be used to characterise a grid and then use subgrids to build source and redistribute the platform-specific binary. The figure illustrates the following three steps:

1. The starting point is an uncharacterized grid.
2. Resources are characterised and grouped into platform-specific subgrids.
3. A representative resource is chosen from each subgrid. This resource performs the compilation and the completed deployment is then archived and redistributed to other resources of that subgrid.

Deploying native software, from source code, across a heterogeneous grid using DistAnt has been demonstrated in a major deployment case study [3]. Using DistAnt, the GAMESS quantum chemistry package and a FORTRAN compiler were installed from source across a testbed of 4 different platform types. A DistAnt workflow was written which characterised the available resources, deployed GAMESS source to the fastest resource of each platform, preprocessed and compiled the source and then redistributed the binary to the whole testbed. If no resources of a particular platform supported a FORTRAN compiler then the gcc-g77 FORTRAN compiler was automatically installed. The deployment was performed with no changes to GAMESS code, very minimal changes to the existing GAMESS build file, and no changes to gcc-g77 code

or build file. The deployment workflow was around 300 lines long and a complete deployment over 8 selected resources took 92 minutes.

This case study demonstrated that a DistAnt workflow could be used to build and deploy software quickly over a uncharacterized grid. In addition, once a deployment workflow was written, it was easily executed, repeatable and non-user-intensive.

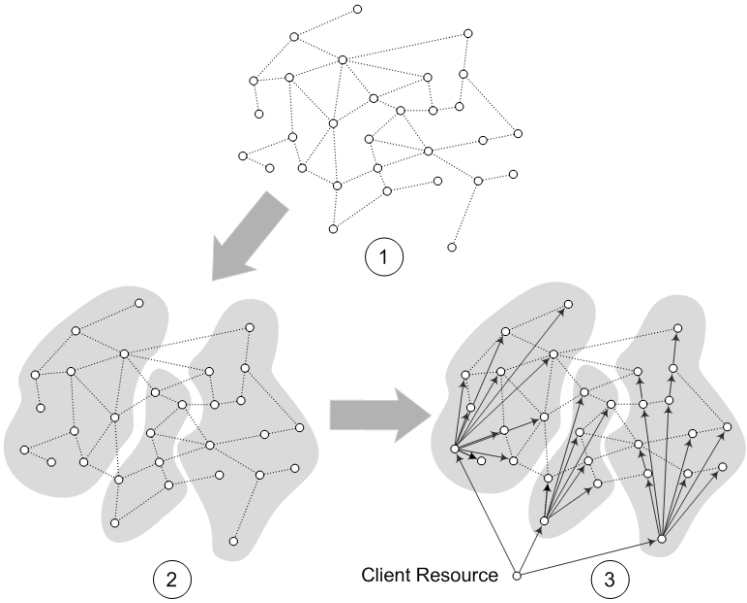


Figure 3. The logical sequence of steps to build source and redistribute binaries using subgrids.

2.5.2. Managing Heterogeneity through Virtualization

Deployment can also be simplified if resources support a common execution environment. In this way, software can be written and built for the virtual environment, rather than a range of specific native platforms, libraries, compilers and environments. Deployment decisions can be made without considering platform and system heterogeneity.

2.5.3. The Motor Application-Runtime

Motor [4] is an application-runtime for HPC, with a runtime-internal communication library. It is an environment based on the Common Language Infrastructure (CLI)[1] [7] and supports two sets of HPC message passing operations:

1. A standard set of MPI-like operations. These provide an efficient message passing mechanism, integrated within the runtime managed memory environment.

[1] The CLI is a Microsoft sponsored specification which defines a language-independent application-runtime environment. It is better known by its commercial implementation, .Net. A range of compilers have been developed which target the CLI environment, including: C#, FORTRAN, C++, Perl, LISP, Java, Visual Basic and C. The SSCLI is the 'shared source' reference implementation of the CLI.

2. A set of higher-level operations for structured data transport. These provide a convenient mechanism to *automatically* transport objects, arrays and object trees.

The Motor implementation is a synthesis of three parts: the Microsoft Shared Source CLI (SSCLI)[1] [12], which provides an existing runtime; MPICH2 which provides a portable MPI implementation; and an additional set of components which implement the higher level message passing operations, and interface to the integrated MPICH2 library. A detailed description of the Motor implementation and its performance is provided in [4]. This paper provides an overview of the Motor architecture because it provides an environment with a guaranteed HPC communication library and is an important part of our deployment framework.

The Motor Runtime-Internal Architecture

A number of environments provide a Java or .Net parallel communication mechanism. These projects can be grouped into two categories:

1. *Pure managed* implementations which execute entirely over the virtual machine. These include JMPI [13] and jmpi [14]. Because they are written entirely in Java they are portable, but are also inefficient.
2. *Runtime-external* implementations which provide a managed interface to an underlying native communication facility. This includes systems such as mpiJava [15], JavaMPI [16], MPIJ [17] and the Indiana .Net bindings [18]. The runtime-external message passing systems perform significantly better than pure managed systems because the high performance communication mechanism has direct native access to transport services.

In both of these classifications the runtime environment is unaware of the HPC communication library. In contrast, the Motor architecture is *runtime-internal*, whereby the runtime environment provides a shared memory or message passing abstraction.

Environments such as Java and the CLI provide web service, user interface, security and networking libraries as *standard* because these are essential for building business and desktop software. We believe that a *high performance* runtime environment design should follow the same philosophy and include essential libraries as standard. Therefore, Motor extends an existing runtime environment with high performance communication, thereby creating a *complete* runtime environment for HPC. The Motor runtime-internal architecture is illustrated in Figure 4, in contrast to the runtime-external architecture.

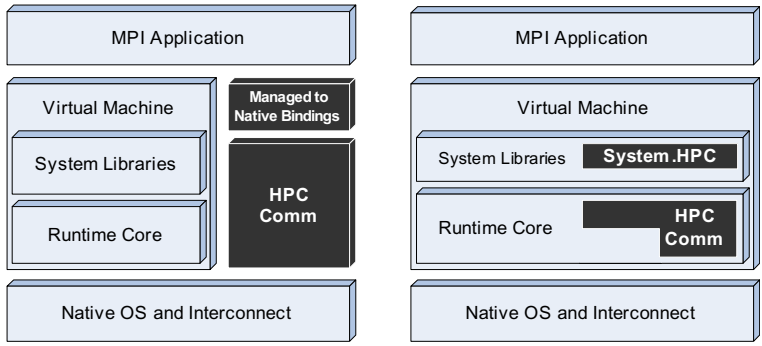


Figure 4. The runtime-external architecture (left), in contrast to the Motor runtime-internal architecture (right).

A *complete* runtime environment is important both from the developer's perspective and is also important for deployment; it is a guarantee that the runtime will support a well defined high performance communication abstraction. In contrast, managed-wrapper MPI implementations must be rebuilt to interface to a resource-specific native MPI implementation. The message passing library is an afterthought, rather than a service guaranteed by the runtime library.

In addition, there are a number of other major reasons why the Motor runtime-internal design is preferable:

1. Runtime-external implementations lack portability – the interface must be rebuilt and sometimes edited for each different underlying native implementation. Therefore, this architecture conflicts with the compile-once-run-anywhere ethos of managed software development. In a runtime-internal implementation, much of the implementation can be defined in higher layers so that only low-level communication primitives and process-management-specific code need to be re-implemented.
2. Implementing the communication abstraction internally results in better performance than the alternative runtime-external architecture. In micro-benchmarks the Motor MPI library has performed noticeably better than runtime-external MPI implementations [4]. For example, in a ping-pong micro-benchmark Motor performs better than runtime-external bindings hosted by the SSCLI using MPICH2; 16% at a peak; 8% on average over all buffer sizes; and 3% on average over buffer sizes greater than 65,536 bytes. These are positive results considering Motor is a synthesis of these same major components: SSCLI and MPICH2 [4]. In addition we have tested Motor using a real application, Shallow Water which is an implementation of the shallow water model [19]. We implemented a C version of Shallow Water which is executed using the Motor environment. In these results [20], Motor performs favorably against the SSCLI, using runtime-external bindings. On average Motor performs 21.9% faster than SSCLI and only 4.5% slower than the native implementation.
3. A runtime-internal implementation has full access to internal runtime services, such as the runtime thread model, memory management, and the object and class model. The advantage of this is that the environment can support higher levels of functionality, such as the Motor structured data transport operations. The result is also performance. In ping-pong micro-benchmarks the Motor structured data transport operations performed significantly better than similar runtime-external MPI implementations using runtime-provided serialization [4].

2.6. The Application Development Lifecycle

Deployment is an integral part of the overall software development, testing and execution lifecycle. Because DistAnt is an extension of Ant, it allows a user to write a deployment workflow as part of the build file and test software on a testbed, during and after the development phase.

It is important that deployed software is easy to execute. Therefore, the DistAnt system has been integrated with a high-level resource broker and scheduling environment, Nimrod/G [21]. This allows users to write regular Nimrod/G execution descriptions and reference a DistAnt deployment using the deployment reference [3].

2.7. Dependency Management

Managing native software dependencies is a challenging task over a large-scale heterogeneous grid. For example, different resources support different libraries versions or compiler implementations. Because the DistAnt Workflow is built using the Ant build file system, it allows the developer to manually define software dependencies in the same way as is provided by Ant.

In addition our framework provides the Automatic deployment Tool (ADT), which automatically generates and enacts a workflow for deploying a Motor executable and its libraries over a testbed. It performs this deployment in two steps. First, it queries Motor binaries recursively requesting dependency information. Because Motor is based on the CLI, it supports self-describing binaries which contain information about library dependencies. Second, it forms and enacts a DistAnt Workflow based on the computed dependency tree, thus deploying the executable and all its libraries over the grid testbed.

However, not every resource will require each binary. Some resources might already support one or more application components, perhaps due to an earlier deployment. Therefore, the ADT generates a workflow which queries resources for the presence of necessary binaries. It groups resources into subgrids which require particular binaries and deploys those binaries only to the required subgrids.

The result of the ADT is that users do not need to manually define library dependencies.

3. Middleware Supported Deployment

Our framework and its components support the following software development, deployment and execution steps (Figure 5):

1. *Software development*. Users have the option of writing managed code for the Motor application-runtime rather than porting native code for a range of platforms or environments.
2. *Software compilation.* Motor application-runtime compiled binaries are:
   - Process-independent, which enables compile-once-run-anywhere deployment. This means that users can compile code once for all resources, rather than once for each platform; and
   - Self-describing, which allows management of library dependencies using the ADT.

   Alternatively, DistAnt supports users to compile native software across a range of platforms by compiling on one of each platform and redeploying the resulting binary.
3. *Define the application testbed.* The user writes a DistAnt workflow which characterises grid resources. Similar resources can be grouped together based on these same attributes using the subgrid construct. Therefore, the user can easily define a testbed by selecting resources which match resource attributes to software and experiment requirements.
4. *Realize the application testbed*. The user realizes the testbed by writing a deployment workflow which deploys the given application. Alternatively, a deployment workflow can be automatically generated based on a runtime executable and its library dependencies.

The deployment service provides facilities to perform installation and configuration procedures on each remote resource. Different remote actions can be performed at different resources by defining subgrids and specific deployment routines.

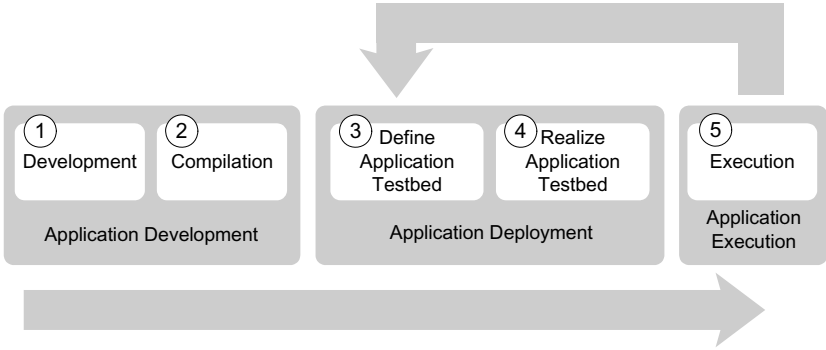


Figure 5. Software development, deployment and instantiation steps using the IDEA framework.

5. *Execution*. A resource management system or higher level grid service, such as Nimrod/G, is used to schedule execution over the testbed. DistAnt deployment references provide an execution handle to the deployment across the entire testbed. The HPC runtime ensures execution.

Importantly, these steps are easily revisited, allowing the user to redefine the testbed and adapt to changes in the physical grid architecture, software requirements or experiment requirements.

4. Related Work

A number of projects have focused on providing a suitable runtime environment for large-scale e-Science applications [21-27]. For example, the Globus project defines middleware interfaces that support application invocation in a secure, distributed, environment [25]. However, none of these environments specifically address deployment over a large-scale distributed and heterogeneous grid.

System configuration and management provide the functionality to transform a blank machine into a functional system [28-30] including many proprietary systems [30]. Similarly, component and object-oriented management systems, allow the user to define components and their relationships in order to automatically generate the described application system [31, 32].

However, these configuration management systems and component management systems are *super-user-oriented* systems; they define a *centralized* mechanism providing administration and configuration over systems within institutional control. This is an evolution of cluster management where deployment is largely a non-issue; users typically only need to deploy software once, on a shared file system, so that it can be executed across the entire cluster. In contrast, deploying software over a loosely coupled set of grid resources requires an individual deployment on each resource. Furthermore, a multi-institutional grid has no one overarching super-user to control software issues such as dependencies, version issues or clashing libraries.

Some grid service hosting and deployment systems, provide a facility to 'hot' deploy grid services [33-39]. None of these systems support deployment of non-hosted software, such as native parallel applications. Nor do they deal with other user-oriented issues such as scale and heterogeneity, which we have identified as significant issues.

In addition, platform-level virtualization systems, provide an infrastructure to host application using a platform-level virtual machine such as VMWare or Xen [40-46]. This approach requires a very large effort to deploy even the simplest of software. For example, to deploy relatively simple code an entire operating system image would need to be generated, configured or transported to the remote resource. Furthermore, platform-level deployment does not address specifics of the operating system, virtual or otherwise.

5. Conclusion

The outcome of our framework is that users are provided with an overall environment which supports the creation of application testbeds.

DistAnt is user-oriented allowing a regular user to deploy a native or managed application over an uncharacterized and heterogeneous set of resources. The DistAnt Service is significant because it defines a common access point and interface for deployment, while the DistAnt Workflow provides a facility to describe and enact the steps to define and realize a testbed.

In addition, Motor provides a common environment for high performance applications. Thus, users can develop and compile application software for the Motor environment rather than a range of platforms, simplifying application development and deployment. The integration of a message passing library guarantees that the runtime will support a well known parallel communication mechanism. Finally, the ADT is significant because it automatically deploys a Motor executable and its libraries.

References

[1] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, 1st edition ed: Morgan Kaufmann Publishers, 1998.

[2] W. Goscinski and D. Abramson, "Distributed Ant: A System to Support Application Deployment in the Grid" *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pp. 436 - 443, 2004.

[3] W. Goscinski and D. Abramson, "Application Deployment over Heterogeneous Grids using Distributed Ant" *First International Conference on e-Science and Grid Technologies (e-Science 2005)*, pp. 361 - 368, 2005.

[4] W. Goscinski and D. Abramson, "Motor: A Virtual Machine for High Performance Computing" *The Fifteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC2006)*, 2006.

[5] D. Abramson, "Applications Development for the Computational Grid" *Lecture Notes in Computer Science*, vol. 3841, 2006.

[6] "Java", 2006, http://java.sun.com.

[7] "Standard ECMA-335: Common Language Infrastructure, 3rd edition (June 2005)", 2005, http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[8] "Microsoft.NET", 2006, http://www.microsoft.com/net/.

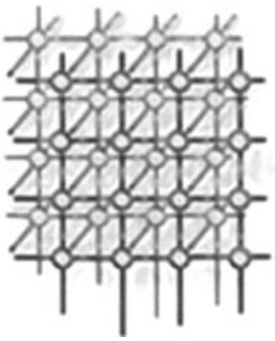
[9] "TeraGrid", 2006, http://www.teragrid.org/.

[10] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A Resource Management Architecture for Metacomputing Systems" *Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[11] "Apache Ant", 2006, http://ant.apache.org/.
[12] "Shared Source Common Language Infrastructure 1.0 Release", 2003, http://msdn.microsoft.com/net/sscli/.
[13] S. Morin, I. Koren, and C. Krishna, "JMPI: Implementing the message passing standard in Java" *International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops.*, 2002.
[14] K. Dincer, "Ubiquitous message passing interface implementation in Java: jmpi" *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, 1999.
[15] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim, "mpiJava: An Object-Oriented Java interface to MPI" *International Workshop on Java for Parallel and Distributed Computing*, 1999.
[16] S. Mintchev, "Writing programs in JavaMPI" *Technical Report MAN-CSPE-02*, 1997.
[17] G. Judd, M. Clement, Q. Snell, and V. Getov, "Design issues for efficient implementation of MPI in Java" *Proceedings of the ACM 1999 Java Grande Conference*, 1999.
[18] J. Willcock, A. Lumsdaine, and A. Robison, "Using MPI with C# and the Common Language Infrastructure" *Concurrency and Computation: Practice and Experience*, 2005.
[19] C. Vreugdenhil, *Numerical Methods for Shallow-Water Flow*: Kluwer Academic Publishers, 1994.
[20] W. Goscinski, "IDEA: An Infrastructure for the Deployment of e-Science Applications" *Monash University PhD Thesis*, 2007.
[21] D. Abramson, J. Giddy, and L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?" *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 520 - 528, 2000.
[22] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids" *Proceedings. 10th IEEE International Symposium on High Performance Distributed Computing*, 2001.
[23] F. Berman, A. Chien, K. Cooper, J. Dongarra, Ian Foster, D. Gannon, L. Johnsson, K. Kennedy, Carl Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, "The GrADS Project: Software Support for High-Level Grid Application Development" *International Journal of High Performance Computing Applications*, vol. 15, 2001.
[24] K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe, "The WS-Resource Framework", 2004, http://www.globus.org/wsrf/specs/ws-wsrf.pdf.
[25] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit" *The International Journal of Supercomputer Applications and High Performance Computing*, 1996.
[26] I. Taylor, I. Wang, M. Shields, and S. Majithia, "Distributed computing with Triana on the Grid" *In Concurrency and Computation:Practice and Experience*, vol. 17(1-18), 2005.
[27] R. Stevens, A. Robinson, and C. A. Goble, "myGrid: Personalised Bioinformatics on the Information Grid" *11th International Conference on Intelligent Systems in Molecular Biology*, 2003.
[28] G. Bruno, P. Papadopoulos, and M. Katz, "Npaci rocks: Tools and techniques for easily deploying manageable linux clusters" *3rd IEEE International Conference on Cluster Computing (CLUSTER'01)*, 2001.
[29] P. Anderson, G. Beckett, K. Kavoussanakis, G. Mecheneau, J. Paterson, and P. Toft, "Large-Scale System Configuration with LCFG and SmartFrog", 2003, http://www.gridweaver.org/WP3/report3_1.pdf.
[30] P. Anderson and A. Scobie, "LCFG: The Next Generation" *UKUUG Winter Conference*, 2002, http://www.lcfg.org/doc/ukuug2002.pdf.
[31] Patrick Goldsack, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray, and P. Toft, "SmartFrog: Configuration and Automatic Ignition of Distributed Applications", 2003.
[32] P. Anderson, G. Beckett, K. Kavoussanakis, G. Mecheneau, and P. Toft, "Technologies for Large-Scale Configuration Management", 2002, http://www.gridweaver.org.
[33] D. Kurzyniec, T. Wrzosek, D. Drzewiecki, and V. Sunderam, "Towards self-organizing distributed computing frameworks: The H2O approach" *Parallel Processing Letters*, vol. 13, 2003.
[34] M. Migliardi, V. Sunderam, A. Geist, and J. Dongarra, "Dynamic reconfiguration and virtual machine management in the Harness metacomputing system" *Lecture Notes in Computer Science*, vol. 1505, 1998.
[35] L. Qi, H. Jin, I. Foster, and J. Gawor., "HAND: Highly Available Dynamic Deployment Infrastructure for Globus Toolkit 4", 2006, http://www.globus.org/alliance/publications/papers/HAND-Submitted.pdf.
[36] J. Weissman, S. Kim, and D. England, "A Framework for Dynamic Service Adaptation in the Grid: Next Generation Software Program Progress Report" *19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, 2005.
[37] J. Weissman, S. Kim, and D. England, "Supporting the Dynamic Grid Service Lifecycle" *5th International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, 2005.

[38] G. Wasson, N. Beekwilder, M. Morgan, and M. Humphrey, "OGSI.NET: OGSI-compliance on the.NET framework" *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, 2004.
[39] E.-K. Byun, J.-W. Jang, W. Jung, and J.-S. Kim, "A Dynamic Grid Services Deployment Mechanism for On-Demand Resource Provisioning" *5th International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, 2005.
[40] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo, "VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing" *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.
[41] K. Keahey, I. Foster, T. Freeman, X. Zhang, and D. Galron, "Virtual Workspaces in the Grid" *11th International Euro-Par Conference*, 2005.
[42] X. Zhang, K. Keahey, I. Foster, and T. Freeman, "Virtual Cluster Workspaces for Grid Applications " *ANL technical report, ANL/MCS-P1246-0405*, 2005.
[43] I. Foster, T. Freeman, K. Keahey, D. Scheftner, B. Sotomayor, and X. Zhang, "Virtual Clusters for Grid Communities" *IEEE International Symposium on Cluster Computing and the Grid*, 2006.
[44] R. Figueiredo, P. Dinda, and J. Fortes, "A Case For Grid Computing On Virtual Machines" *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.
[45] S. Adabala, V. Chadha, P. Chawla, R. J. O. Figueiredo, J. A. B. Fortes, I. Krsul, A. M. Matsunaga, M. O. Tsugawa, J. Zhang, M. Zhao, L. Zhu, and X. Zhu, "From virtualized resources to virtual computing grids: the In-VIGO system" *Future Generation Computer Systems*, 2005.
[46] A. Ganguly, A. Agrawal, P. Boykin, and R. Figueiredo, "WOW: Self-Organizing Wide Area Overlay Networks of Virtual Workstations" *Proceedings of the Fifteenth International Symposium on High Performance Distributed Computing (HPDC-15)*, 2005.

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2008)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1325



Parallel Programming on a High-Performance Application-Runtime

Wojtek James Goscinski*,† and David Abramson

Faculty of Information Technology, Monash University, Vic. 3145, Australia

SUMMARY

High-performance application development remains challenging, particularly for scientists making the transition to a heterogeneous grid environment. In general areas of computing, virtual environments such as Java and .Net have proved to be successful in fostering application development, allowing users to target and compile to a single environment, rather than a range of platforms, instruction sets and libraries. However, existing runtime environments are focused on business and desktop computing and they do not support the necessary high-performance computing (HPC) abstractions required by e-Scientists. Our work is focused on developing an application-runtime that can support these services natively. The result is a new approach to the development of an application-runtime for HPC: the Motor system has been developed by integrating a high-performance communication library directly within a virtual machine. The Motor message passing library is integrated alongside and in cooperation with other runtime libraries and services while retaining a strong message passing performance. As a result, the application developer is provided with a common environment for HPC application development. This environment supports both procedural languages, such as C, and modern object-oriented languages, such as C#. This paper describes the unique Motor architecture, presents its implementation and demonstrates its performance and use. Copyright © 2008 John Wiley & Sons, Ltd.

Received 16 April 2007; Revised 18 December 2007; Accepted 10 February 2008

KEY WORDS: parallel programming; virtual machines; message passing

1. INTRODUCTION

Parallel programs are most commonly developed natively[‡]; they are written and compiled directly for a specific operating system and instruction set. This approach offers the highest level of

*Correspondence to: Wojtek James Goscinski, Faculty of Information Technology, Monash University, Vic. 3145, Australia.
†E-mail: wojtek.goscinski@infotech.monash.edu.au

‡The code developed for a virtual machine is often referred to as *managed code*, which means that it is managed by an underlying runtime environment. This is in contrast to regular *native code*, which interfaces directly to the operating system.

Copyright © 2008 John Wiley & Sons, Ltd.



performance and allows the programmer full control of memory management. In contrast, the current approach in software engineering has been to move toward a 'managed' (see Footnote ‡) virtual machine architecture, such as Java or the common language infrastructure (CLI) [1], and its commercial implementation, Microsoft .Net. These environments provide a common runtime environment, regardless of the underlying platform, enabling compile-once-run-anywhere application deployment. They define a range of features and services, such as a common-type system, memory management, portable binary format, self-describing binaries and security. In addition, they provide a common set of libraries—a standard set of functionality, which is likely to be required by the user. In the case of Java and .Net, this includes web services, IO, database, user interface, security and networking functionality. Most significantly, runtime environments provide an *execution guarantee*. In essence, a runtime specification defines a contract between a user code and the runtime. Thereby, the code is guaranteed to be executed on a correctly implemented runtime, so long as both parties adhere to the contract.

Taking a similar approach for high-performance computing (HPC) would provide users with a *virtual HPC environment*, which would support the functionality and services that are important to high-performance application developers. For example, message passing, shared memory, process management and associated mathematics libraries, such as linear algebra libraries, are all important tools that should be considered for inclusion into a runtime for high-performance application development. Thereby, users would be provided with a common environment that would make high-performance applications easier to write and more accessible to scientists.

Such an environment is particularly valuable now that HPC is adopting grid computing infrastructure [2], where users have access to a range of heterogeneous resources. Ideally, an e-Scientist, with access to a grid of resources, should be able to pick the resource that best suits their needs. However, realistically, they are often constrained by deployment issues such as platform and environmental dependence. The Message Passing Interface (MPI) specification simplifies scientific application development because it defines a single interface that is implemented over a wide range of platforms and interconnects. However, high-performance application developers still face an assortment of non-MPI porting and deployment issues such as platform-specific system calls, recompilation and library problems. Native code requires porting for different platforms, which is a major undertaking considering the potential complexity of e-Science software and the range of platforms that might constitute a grid. Furthermore, native code needs to be recompiled for each specific platform, which itself is a major effort. This limits users when making deployment and execution decisions. Writing highly tuned, complex scientific applications, for a range of heterogeneous resources remains difficult and time consuming [3–6].

An application-runtime[§] allows users to target and compile to a single environment, rather than a range of platforms, instruction sets and libraries. However, existing runtime environments alone are inappropriate for HPC; neither Java nor the CLI provides necessary high-performance functionality, such as the tools and interfaces mentioned above. Nevertheless, researchers are interested in using

[§] To avoid confusion, this paper avoids using the overloaded term 'virtual machine' when referring to application-runtime environments such as Java and the CLI. It is important to note that many virtual machines are not application-runtimes. For example, Parallel Virtual Machine (PVM) is called a virtual machine but it is not an application-runtime environment under a widely accepted definition because it does not provide a common and complete execution environment for application binaries (such as that provided by .Net and Java).

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

runtimes for HPC and significant work has been carried out in this area [7–18]. In particular, a number of projects have proposed a high-performance communication extension to an existing runtime [7,9–15,17], including the Indiana University .Net bindings [17], MPJ [12], mpiJava [11] and work supported by the Java Grande Forum [18]. The most efficient of these systems have all proposed an external addition to a runtime, thereby supporting high-performance communication externally. Using this approach, the runtime and high-performance communication library remain separate. The library interfaces directly to the native operating system and is unable to access virtual machine services. In turn, the virtual machine does not trust the message passing library, treating it as an unsafe native code.

In contrast, the goal of our research is to develop a *new* virtual environment for HPC by integrating the requirements of HPC developers *within* the runtime. The first steps toward this goal are to develop an environment that supports high-performance message passing. In our architecture, the high-performance message passing library exists alongside other virtual machine services such as memory and process management, IO, security and networking (Figure 1). To demonstrate our approach, we provide a proof-of-concept implementation in the form of Motor (MPI + Rotor) [4]. Motor has been developed by integrating a high-performance message passing library, the MPICH2 [19] MPI library, within an existing runtime, Microsoft's Shared Source Common Language Infrastructure (SSCLI) [20,21] (also known as Rotor).

Motor supports both procedural and object-oriented languages. In addition to providing a standard MPI library for procedural languages, Motor provides two MPI-like libraries for efficient object-to-object transport and structured data transport. Because it has been designed to support object-oriented languages, Motor provides functionality not available in MPI, including the ability to transport arbitrary objects, trees of objects and arrays of objects, and the ability to scatter and gather arrays of objects.

Motor is significant for a number of reasons. First, it allows users to write message passing applications without regard for a particular operating system and interconnect. In contrast to native MPI applications, Motor applications are compile-once-run-anywhere. Therefore, HPC developers have simple deployment and guaranteed execution regardless of underlying heterogeneity. Second, applications written for Motor are provided with runtime services and a guaranteed standard set

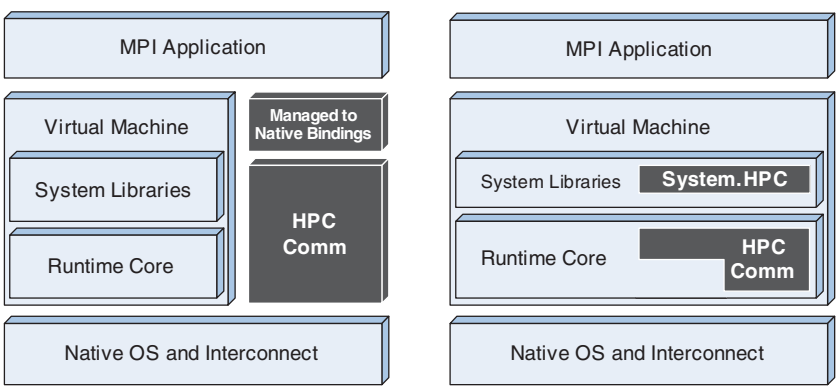


Figure 1. The external architecture (left) in contrast to the Motor-embedded architecture (right).

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



of System libraries, including message passing. HPC developers can develop against a known and standard environment. Third, Motor is language neutral. This means that it can be used for modern object-oriented languages and procedural languages. In particular, Motor is a modern runtime that can be used to execute legacy applications written in C or FORTRAN.

Both architecturally and practically, implementing high-performance message passing directly within the virtual machine has a number of advantages over the existing external approach:

- The message passing library is a *guaranteed* standard subset of the virtual infrastructure. This follows the same philosophy as Java and the CLI, which include essential business-oriented libraries as standard.
- The message passing library has intimate, efficient access to the virtual machine infrastructure, including memory management and the runtime object model. In the case of object-oriented languages, this means that it is able to achieve strong performance while protecting the integrity of the runtime object model.
- Using a layered approach, Motor is portable. Higher-level library functionality is completely portable with the virtual runtime, whereas lower-level functionality can be re-implemented for specific platforms and interconnects.

Finally, Motor is unique as an MPI implementation. Although the managed-wrapper architecture has been implemented several times, we could find no major effort to integrate a high-performance message passing library *directly within* a virtual runtime.

This paper discusses previous work in the area of HPC using virtual machines (Section 2), introduces our architecture for a high-performance virtual machine (Sections 3 and 4), introduces the Object Passing Interface (OPI), an object-oriented MPI (Section 5), introduces relevant technology (Sections 6 and 7), discusses the implementation of Motor (Section 8) and presents performance results (Section 9).

2. HPC OVER VIRTUAL MACHINES

There are a number of approaches to developing a virtual environment for the heterogeneous grid. These include virtualizing the environment at the classical virtual machine level (e.g. VMWare or Xen), which has been investigated by a number of projects [22–29] and at the subsystem component level (e.g. open runtime environment (OpenRTE) [30] and Harness [31]). None of these attempt to virtualize grid resources at the application-runtime level (e.g. Java or .Net), which has proved to be a successful approach for business and desktop applications. Our work is focused on application-runtime virtualization and therefore our review of earlier work is based around projects that have focused on this type of virtualization.

Application-runtimes such as Java and the CLI are characterized by a number of features, including a virtual runtime that just-in-time (JIT) compiles a processor-independent intermediary language (IL) and a standard set of System libraries. In addition, most common runtimes support memory management, which collects abandoned memory and an object-oriented memory model. The CLI is an example of a runtime specification that supports both procedural and object-oriented languages.

In a business and desktop computing environment these features provide an environment for easy prototyping and development. However, in exchange for platform independence and other

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

runtime services, an overhead on performance is created. This paper does not specifically benchmark application-runtimes, however, one study comparing high-performance benchmarks executed over different versions of the CLI and Java can be found in [16]. Although virtual machines do introduce a performance overhead, we believe that the potential advantages in simplifying application development could help foster grid computing within the general science community.

To date, both Java and the CLI focus on desktop, business and mobile applications. Neither explicitly supports HPC abstractions such as high-performance message passing.

2.1. MPI implementations for virtual environments

A number of environments have been developed, which provide a Java or .Net parallel communication mechanism. These projects have either made an argument for using application-runtime environments for HPC or evaluated the relevance of these environments for this task. In addition, this range of projects has demonstrated that the motivation to develop specialized application-runtime environments exists in the high-performance community. Our work continues this research though we have applied a significantly different approach.

mpiJava [11] is a Java wrapper to an underlying native MPI implementation. The mpiJava bindings are based on the MPJ application programming interface (API) (discussed below). mpiJava supports transport of simple-type arrays or objects. Object transport is implemented by serialization using a standard Java serialization. JavaMPI [15] provides an automatically generated wrapper to an underlying native MPI library. Both mpiJava and JavaMPI use the Java native interface (JNI), which provides a Java mechanism to call a native code. JMPI [10] is a pure Java implementation of a subset of MPI. Communication in JMPI is implemented over Java Remote Method Invocation. jmpi [14] is another pure Java implementation, built over the Java sockets library.

The MPJ API [12] is an API specification for Java MPI bindings. Developed by the Message Passing Working Group of the Java Grande Forum [18] it was produced to create a standard Java MPI-like API specification. It is not an official binding to MPI, but it is the most significant attempt to formalize such a binding. MPJ describes a Java-oriented adaptation of the official C++ object-oriented bindings.

The CLI (and its commercial implementation .Net) is more recent than Java and has been the subject of one major effort to create an MPI library. The Indiana University .Net bindings [17] provide two bindings to the underlying native MPICH MPI library: C# bindings, based on the official C++ MPI bindings and MPI.NET, a higher-level interface specifically focused toward the CLI environment. The Indiana bindings use the CLI P/Invoke (Platform Invoke) interface to invoke the underlying MPI library. P/Invoke is similar to JNI; it allows managed CLI code to call native code. The Indiana bindings impose a slight overhead over the native MPICH, but suffer due to the overhead of object pinning [17].

These projects discussed can be grouped into two categories:

- Pure managed implementations of MPI that execute entirely over the virtual machine. These implementations are portable but suffer from inefficiency.
- Managed wrappers that provide a managed interface to an underlying and external native MPI implementation. These systems perform significantly better than pure managed systems because the high-performance communication mechanism has direct native access to transport services.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



The common characteristic of these two classifications is that the runtime environment is unaware of the HPC communication library. In contrast, the Motor architecture is *embedded*, whereby the runtime environment provides a shared memory or message passing abstraction.

Efficient MPI implementations require direct access to the underlying operating system or interconnect, which a pure Java or .Net implementation is unable to provide. Implementing MPI bindings as a wrapper to an underlying native MPI is efficient but introduces a number of architectural issues that are also relevant to our embedded implementation. These include portability, managed-to-native interface, object model integrity and interaction with managed memory (some of these issues have been raised earlier [32]).

Managed wrappers lack portability; the managed wrapper must be rebuilt and sometimes edited for each different underlying MPI implementation. This architecture conflicts with the compile-once-run-anywhere ethos of managed software development and has been raised in an earlier publication [32].

Using a managed-to-native call mechanism such as JNI [33] or P/Invoke [34] imposes an overhead on each MPI call because both JNI and P/Invoke require marshaling and impose security mechanisms.

Object model integrity and interaction with managed memory are discussed in Section 5.

To achieve the highest possible performance, MPI implementations cooperate with underlying low-level services provided by the operating system or transport services provided by the interconnect. In a virtual environment, the runtime provides an extra layer of services, for example, managed memory. However, the MPI implementations that we have discussed are oblivious to the services provided by the virtual machine, instead interfacing directly with the operating system. Circumventing the virtual machine functionality is neither elegant, and as our implementation demonstrates, nor is it more efficient than integrating directly within the virtual machine.

In addition to these architectural problems, all the described systems target a specific language. A number of runtimes have been developed, which provide a common environment for a range of languages, including the CLI and Parrot [35].

3. INTRODUCTION TO MOTOR

Motor is a synthesis of the Microsoft SSCLI and the MPICH2 [19] MPI library.

A number of runtime environments—including various versions of the CLI, Java and Python—were considered for our implementation. We decided to use an implementation of the CLI primarily because it is the predominant language-neutral environment. CLI compilers exist for procedural languages such as C, FORTRAN and Pascal, and object-oriented languages such as C#, Java and Python.

In addition, the CLI is considered a better choice over Java for scientific computing because it supports true multidimensional arrays such as those in C and FORTRAN. In contrast, Java uses the arrays-of-arrays model, which means that in Java n-dimensional arrays are implemented as one-dimensional arrays, with $n - 1$-dimensional array elements. Transporting a multidimensional array constructed using the arrays-of-arrays model is difficult because the array is constructed of multiple objects. Each of these objects could be placed at a different location in memory. Therefore, the array needs to be transported in multiple messages, or it needs to be packed or serialized before

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

transport. In contrast, a true multidimensional array can be transported in one message because it exists at one location.

Both the open source Mono and the SSCLI were considered for this work. The SSCLI performs the worst of all benchmarked CLI implementations [16]. However, it was chosen as the basis of Motor because it is a comprehensive CLI implementation, which by all accounts is very similar to the optimized commercial .Net [21]. Using the SSCLI as the basis of our prototype provides the best insight into developing Motor using a highly tuned commercial runtime such as Java or .Net. The SSCLI is described in Section 6.

The choice of the MPICH2 library was influenced primarily by its layered design, allowing it to support different platforms and interconnects. In addition, MPICH2 is ported to the Windows operating system, providing a starting point for our integration into the SSCLI. MPICH2 is described in Section 7.

In contrast to the managed-wrapper MPI implementations that we discussed, the Motor message passing library has been implemented directly within the virtual machine. The message passing library has been implemented in the same manner as the System libraries in SSCLI and commercial .Net. It consists of a managed library located in the `System.MP` namespace, which interfaces to the Message Passing Core (MPC), within the virtual runtime. The MPC is based on the Windows MPICH2 code-base, ported to the SSCLI Platform Adaptation Layer, which defines a virtual subset-Windows API.

4. LOGICAL DESIGN

Motor provides two sets of operations for parallel communication, supporting both procedural and object-oriented languages:

(1) A standard MPI implementation for procedural languages. This is a language-neutral interface for parallel programming on a distributed memory system.
(2) A set of MPI-like operations for object transport in object-oriented languages. These operations, which are called the Motor Object Passing Interface (OPI), do not directly implement the MPI specification. Rather, they are a set of MPI-derived operations, which are based on the MPI-2 C++ bindings and object model. However, the OPI is specifically designed for an object-oriented environment and supports two types of operations:

- A set of regular object-to-object transport operations, which are adapted to preserve object model integrity in a strongly typed object-oriented environment.
- A set of higher-level operations for structured data transport in object-oriented languages. These operations are called the Motor Structured Data Transport Operations[¶] (SDTO) and they provide automatic serialization and buffer management.

The Motor MPI library is an embedded implementation of the MPI standard. Therefore, we do not discuss the design of this interface in detail. However, the OPI is a new interface design, which

[¶]These operations are called the Extended Object-Oriented Operations in an earlier publication [4].

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

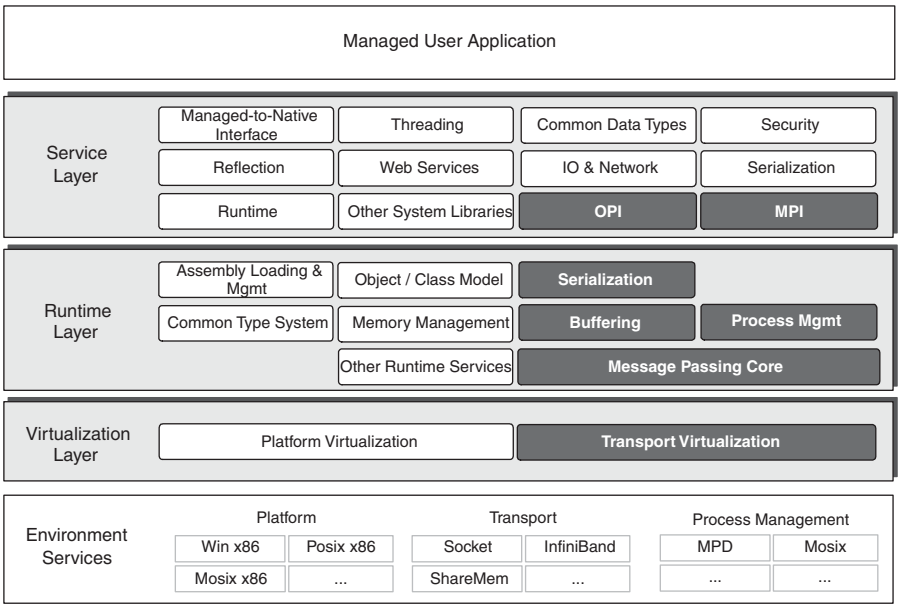


Figure 2. The Motor architecture. Extensions to the architecture of the regular CLI virtual environment are accentuated in black.

we have developed specifically for an object-oriented environment. Therefore, the OPI design is discussed in detail in Section 5.

Figure 2 illustrates the architecture of the Motor virtual machine. Applications are written in a procedural language such as C or FORTRAN or an object-oriented language such as C#. These applications are compiled to the CLI IL. They interface to the runtime through the System library, which includes the above-mentioned message passing libraries. The System MPI library is similar to the earlier mentioned managed-wrapper MPI implementations. However, this library interfaces directly to the MPC, an MPI library implemented within the underlying runtime.

The MPC implements the non-transport-specific parts of the message passing library. It is implemented over a virtualization layer that provides transport and platform virtualization. This virtualization layer provides portability; the Motor runtime can be redeployed with different transport or platform interfaces. For example, implementing a shared memory implementation of MPI would require the addition of a shared memory transport mechanism. Placing the message passing library within the runtime gives it access to internal runtime resources. Particularly, Motor interacts with the garbage collector and the runtime object and class model. The manner in which Motor interacts with these services is presented in Section 8.

5. THE OPI DESIGN

The Motor OPI operations are based on the MPI specification and provide efficient object-to-object transport (Section 5.1) and structured data transport (Section 5.2).

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

5.1. Object-to-object transport

Object-to-object transport means that the developer provides a send object and a receive object. The operations transport object data, whereas internal object headers are not transported. Figure 3 illustrates Motor MPI operations, including selected communicator operations (i) blocking and immediate point-to-point operations (ii) selected collective operations such as broadcast and scatter/gather (iv) and selected MPI-2 functionality, such as dynamic process management (v).

The OPI deviates from the MPI specification to protect memory model integrity. Sections 5.1.1 and 5.1.2 introduce the importance of protecting object model integrity and discuss object-to-object

```
public class Comm
{
   static public Comm World ;                                                    i
   static public Comm Self ;
   static public Comm Null ;
   static public Intercomm GetParent ()
   ...
   public int GetRank ()
   public int GetSize ()
   ...
   public void Recv  (object obj , int source , int tag, Status status)         ii
   public void Recv  (object obj , int offset , int numcomponents , int source ,
      int tag , Status status )
   public void Send (object obj , int dest , int tag)
   public void Send (object obj , int offset , int length , int dest , int tag)
   public void IRecv  (object obj , int source , int tag , Request request )
   public void ISend (object obj , int dest, int tag, Request request )
   ...
   public object ORecv  (int source , int tag , Status status )                 iii
   public void OSend (object obj , int dest, int tag)
   ...
}

public class Intracomm  : Comm
{
   public void Barrier ()                                                        iv
   public void Bcast (object buffer , int root )
   public void Gather (object sendbuf , object recvbuf , int root)
   public void Scatter (object sendbuf , object recvbuf , int root )
   ...
   public Intracomm Spawn (string command , string [] argv,                     v
      int maxprocs , Info info , int root )
   ...
   public object OBcast (object buffer , int root)                              vi
   public object OScatter (object sendbuf , int root )
   ...
}
```

Figure 3. A subset of the Motor OPI System.MP object model. Structured data transport operations can be distinguished by an O prefix.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



transport in a memory-managed environment. On the basis of this discussion, Section 5.1.3 presents a detailed description of these operations.

5.1.1. Object model integrity

Message passing specifications, such as MPI, assume a linear view of memory. For example, MPI specifically defines *buffer-to-buffer* message transport. The user provides three major parameters: a buffer that is read or written to by the underlying data transport mechanism, a count that defines the number of elements transported and a data type that defines the size of each element. This means that the developer defines the memory starting point and length.

However, in an object-oriented model, memory is a managed graph of objects and the managed code has no regular safe access to the underlying linear view. Therefore, buffer-to-buffer transport is not possible because the developer cannot explicitly provide a memory location. Furthermore, as we will discuss, allowing the developer to explicitly define a memory length would be dangerous to the integrity of the memory model and the overarching environment. Owing to these differences, we refer to buffer-to-buffer transport in an object-oriented environment as *object-to-object* transport, where the user provides both a send object and a receive object. The remainder of this section discusses problems with implementing an MPI object-to-object transport in a runtime environment. These are introduced so that we are able to define a Motor MPI-like interface that provides an efficient zero-copy transport while functioning safely within the rules of the runtime environment.

Object-to-object transport is only possible when transporting single objects (including single structures and arrays of simple types because these will be typically located at one location in the memory). Transporting multiple objects using a single object-to-object operation is impossible because multiple objects will most likely be located at different locations in the memory. Likewise, arrays of objects and trees of objects are not located at a single physical memory location and therefore transporting them as a single transport is impossible. Even if they were located at a single location in the memory, objects would be separated by an object header, which should not be transported to another runtime instance.

Implementing the MPI interface as specified, and without any buffer boundary checks or restrictions, allows the user code to inadvertently (or purposefully) destroy the integrity of the object model. This is because in a strongly typed object-oriented memory model, object references are guaranteed to be either null or reference an object of the correct type. Therefore, compromising the integrity of the object model can occur in two ways:

- By overwriting the end of an object, corrupting the object header and object data of the next object in the memory. This would occur if the developer provided a memory length greater than the receiving object.
- By overwriting an object reference with data. This would occur if a receiving object referenced other objects and those references were overwritten by data during the receive operation.

Corruption of the underlying object tree *will* crash the application environment. This is because the garbage collector requires object model integrity to successfully perform its service. These two occurrences are also a problem in a linear static memory model where it is the responsibility of the developer to maintain data integrity. However, in a managed environment, it is the responsibility of

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

the *runtime* to guarantee the integrity of the memory model. It would be counter-intuitive to allow the embedded message passing library to compromise object model integrity.

5.1.2. Object-to-object transport with managed memory

Efficient message passing implementations interface directly to an underlying data transport mechanism, whether it is sockets, shared memory or a proprietary interconnect. The underlying transport mechanism reads and writes data directly to and from the supplied memory buffer. This is known as a zero-copy message transfer and it is efficient because it avoids extraneous memory copies. However, this principle interferes with garbage collection.

Several approaches and algorithms for garbage collection have been developed [36]. Environments such as Java and .Net employ generational collection where new objects are allocated in the younger generation and then if they pass a garbage collection, they are promoted to the elder generation. In essence, generational garbage collection applies a heuristic that exploits the fact that objects have varying lifetimes. Objects in the youngest generation have a low survival rate, whereas objects in the elder generation have a higher survival rate. Therefore, the younger generation is collected often, whereas the elder generation is collected less frequently. When a set of objects are promoted to the elder generation, they are copied to the elder generation and compacted to reduce fragmentation.

Implementing an efficient zero-copy MPI in any managed memory environment that has the power to move memory is problematic because the memory might be moved or collected during an MPI operation. Therefore, the solution is to pin the object while transport is underway. Pinning is necessary, but undesirable. It interferes with the garbage collector's standard behavior and causes memory fragmentation. Furthermore, each pin/unpin operation imposes a small performance overhead. For example, in [17] the Indiana CLI MPI bindings suffer from memory pinning.

We believe that relying on the user to pin and unpin memory buffers is dangerous because failing to unpin a memory buffer results in leaking memory. Therefore, we suggest that the runtime should provide automatic pinning and unpinning. Automatically providing memory pinning is relatively simple for blocking MPI operations: the object is pinned at the start of the operation and unpinned once the operation is completed. However, non-blocking operations require unpinning later, once the underlying transport operation has actually been completed. The MPI.NET description in [17] does not illustrate any mechanisms to automatically unpin buffers for non-blocking operations.

Motor provides automatic pinning and unpinning of managed send and receive objects. The manner in which this is achieved is presented in Section 8.3.2.

5.1.3. Motor OPI object-to-object interface

The Motor OPI object-to-object interface is based on the official C++ MPI interface, providing the same basic underlying functionality and class model (Figure 3). However, individual OPI operations have been modified to protect the integrity of the memory model. The changes are as follows:

(1) The provided buffer is a single object (arrays of simple types are allowed because they are a single object). The *count* parameter will always be 1 and has been removed.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



(2) Only object types with no object references may be used as send or receive objects. This prevents overwriting references. Passing objects with references is only allowed through the SDTO.
(3) Object type is easily determined in a strongly typed object-oriented environment and therefore the data type (MPI_Datatype) parameter has been removed.
(4) Transporting portions of objects or offsetting into an object is not supported because in most object-oriented environments there is no safe way to refer to a subset of an object.
(5) However, transporting array subsets is supported. An overloaded set of operations cater for array transport and include an offset and count parameter.

These changes protect the integrity of the underlying object model while retaining a high-performance zero-copy transfer. The effect of these changes is discussed in Section 9.2.1, which describes porting an existing scientific application to the Motor runtime.

5.2. The Motor SDTO

The Motor SDTO are a set of higher-level operations that extend the OPI library to transport structured data, including objects, array of objects and entire object trees. They provide automatic serialization, de-serialization and buffer management. These operations do *not* provide a zero-copy object-to-object transport. Rather, the sender provides a send object that is serialized to a buffer and transported. The recipient receives to a buffer, deserializes objects from the buffer and returns these to the user.

The Motor OPI design separates structured data transport operations because they perform fundamentally different types of transport: OPI operations are efficient zero-copy operations; whereas the SDTO are slower, using serialization. However, the SDTO provide the ability to transport a wider range of structured data. Figure 3 shows a range of SDTO, including regular send/receive operations (iii); and collective communications, including broadcast operations and scatter/gather operations for arrays of objects (vi).

Section 5.2.1 introduces structured data transport, whereas Section 5.2.2 describes structured data transport in detail, focusing on the interface and the serialization and buffering mechanisms.

5.2.1. Structured data transport

Because the MPI specification assumes a linear view of the memory, it does not take advantage of the structural and self-describing nature of an object-oriented memory model. For example, MPI provides pack and unpack routines for the transport of structural data, which must be used by the developer explicitly. In an object-oriented environment, structural data can be transported automatically by serializing a subset of the object model and then transporting the serialized representation. At the receiving end, the flat representation is deserialized. Because object transport requires a serialization mechanism, it does not achieve the same level of performance. However, it is a significant feature in terms of usability; developers can *automatically* transport structured data and avoid data packing and unpacking operations.

A number of MPI implementations (including mpiJava and the Indiana MPI .NET bindings) have discussed using the standard runtime-provided object serialization mechanism for object transport

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

(Section 2.1). However, standard serialization mechanisms produce a single atomic flat representation, which cannot be split or offset similar to the standard memory. Therefore, implementing scatter and gather operations is difficult. For example, to scatter an array of objects over N hosts, the MPI library would need to create N new subarrays and serialize them individually. An embedded passing implementation is at a significant advantage to provide a fast object transport mechanism because it can support a fast custom serialization mechanism. In addition, a custom serialization mechanism can be designed to create a split array representation, which can be used for scatter and gather routines.

5.2.2. Motor OPI structure data transport interface

The SDTO (shown in Figure 3, distinguished by an 'O' prefix) are designed to transport:

- Single objects: transporting a single object is the default behavior and any object can be transported. This operation transports an object's data and avoids transporting referenced objects, which are replaced with null.
- Arrays of objects: an array is treated differently from a normal object. By default an array is transported together with the array-entry objects it references. An overloaded set of operations provide an offset and count parameter to transport subsets of arrays.
- Trees of objects: to transport a tree of objects this interface defines a declarative attribute[‖], *Transportable*. This attribute is used to decorate object references that should be propagated during the operation.

Figure 4 shows an example of a linked list element that uses the Transportable attribute to denote references to be propagated during transport. In this case, the buffer object and next will be propagated, next2 will not.

Runtime serialization mechanisms also support a declarative attribute that defines reference propagation. A key distinguishing feature of our Transportable attribute is that it is an *opt-in*

```
class LinkedTree
{
       [Transportable] public int[] buffer;
       [Transportable] public LinkedArray next;
       public LinkedArray next2;
}
```

Figure 4. An example of the use of the Transportable attribute to annotate references for serialization. This example uses C# syntax.

[‖]Declarative attributes are declarations added to the code, which can be queried at runtime, allowing the developer to describe and query type information. 'Custom attribute' is the CLI name for these declarative metadata mechanisms, whereas in Java they are termed as 'annotations'.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



mechanism. Users specify the references that they want to be propagated. On the other hand, reference propagation attributes in Java and the CLI require users to specifically opt fields out of propagation. We prefer an opt-in mechanism to limit the amount of data users are unwittingly sending and receiving.

Motor implements a custom internal serialization mechanism to provide serialization for structured data transport. Existing runtime environments already provide a serialization mechanism. However, these mechanisms are comprehensive. For performance, the Motor serialization mechanism can be built to include the minimum information necessary to transport an object. In addition, implementing scatter and gather operations requires a mechanism to split the serialized representation. Standard serialization mechanisms produce a single atomic flat representation, which cannot be split or offset similar to the standard memory. The Motor serialization mechanism automatically creates a split representation.

Structured data transport requires a buffer for serialization, which introduces two issues. First, continually allocating buffers for serialization could result in a large overhead on performance. Second, memory-bound applications require diligent user management of memory, including memory buffers. However, some users would prefer automatic buffer management.

In response, the Motor design proposes two approaches to buffering for object transport serialization:

- The user can manage their buffering using regular OPI routines. The Motor custom serialization mechanism is exposed to the user, allowing them to explicitly prepare a buffer, serialize the object tree and perform transport, using regular OPI routines.
- The SDTO provide *automatic* buffer management. Buffers are automatically created when required and stored in a stack for later use. At garbage collection, the stack is checked for buffers that have been unused since the last garbage collection and these are unallocated, freeing unused memory.

6. THE SSCLI

Microsoft's SSCLI includes the CLI runtime, base libraries, a JIT compiler, a C# compiler and a Jscript compiler [20,21]. It is the reference implementation of the CLI and supports both procedural and object-oriented languages. The next section provides an overview of relevant parts of the SSCLI. Figure 2 provides a diagram of the original SSCLI, with our additions in bold. The object-oriented and managed memory aspects of the SSCLI are described in detail because they are relevant to our OPI implementation.

6.1. System library and the CLI runtime

Applications are written for the CLI interface with the System library, which is managed library, written in C#. The System library interfaces to the underlying CLI runtime.

The CLI runtime provides memory management, JIT compilation, the common-type system and library loaders. In addition, much of the functionality offered by the System library is actually implemented by the CLI runtime. This is because the functionality offered by the System library requires intimate interaction with the underlying object model and runtime services, which is only

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

available from within the runtime. Likewise, performance critical functionality of the System library is implemented in the underlying runtime because the C++ runtime can achieve higher performance than the managed System library.

The interface between the System library and the underlying virtual runtime is implemented using an internal call mechanism. This mechanism is declared in the System library using the InternalCall attribute. Within the runtime, this call mechanism is called an FCall (Fast Call), which is a trusted call. FCalls are more efficient than the regular managed-to-native P/Invoke calls because they do not have parameter marshaling and security checks.

FCalls have a number of peculiarities that make them difficult to implement. First, their parameter ordering must match the call ordering of the JIT calling convention. Therefore, they are declared and implemented using a set of macros that define the correct parameter ordering. Second, they must behave similar to a managed code. This means that they must periodically yield to the garbage collector, in case garbage collection is necessary. If yielding is not performed and a garbage collection is required, the FCall would make all other threads wait until it polls for collection. Only when *all* threads enter the safe state does collection commence. Third, unlike in the managed code, the runtime cannot and does not keep track of object pointers in an FCall. Therefore, it is the programmer's responsibility to protect object pointers by declaring them using a set of provided macros. When writing FCalls it is essential to understand under which circumstances garbage collection can occur and which object pointers need to be protected. Otherwise, object pointers will be inadvertently trashed when the objects are moved or collected during garbage collection.

6.2. The garbage collector

The SSCLI virtual runtime provides a two-generational garbage collector. Objects are originally allocated in the younger generation and if they pass a garbage collection, they are promoted to the elder generation. Generational garbage collection exploits the fact that different objects have different lifetimes. Objects in the youngest generation have a low survival rate, whereas objects in the elder generation have a higher survival rate. Therefore, the younger generation is collected often, whereas the elder generation is collected less frequently. When a set of objects are promoted to the elder generation, they are copied to the elder generation, with compaction to reduce fragmentation. Once in the elder generation, objects are collected if abandoned, but are no longer compacted.

The garbage collector maintains a list of objects that require pinning and these objects are not moved. However, automatic pinning has a two-fold negative effect and is undesirable because:

(1) Continually requesting memory pinning for each message passing operation adds a small performance overhead.
(2) Pinned objects disrupt the *normal* ideal operation of the garbage collector and cause fragmented memory.

Garbage collection occurs when remaining memory on the heap runs low. Specifically, it is triggered by a request for a new object. To perform a garbage collection, all threads must be frozen in a safe point. To facilitate this, the JIT-compiled code periodically polls, to yield itself to garbage collection, in case it is necessary.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



6.3. The runtime object and class model

Every SSCLI object is an instance of the most basic type, `System.Object`, and exists on the garbage-collected heap. Within the SSCLI runtime, `System.Object` is defined by the internal type, `Object`, which contains just one field, a reference to the object's `MethodTable`. All of an object's instance data exist directly after the `MethodTable` reference.

The `MethodTable` is the gateway to commonly accessed type information. This includes a reference to an array of `FieldDesc` entries. Each field of every class type is described with a `FieldDesc`; an optimized structure, using a bit field to describe field information.

In addition to the runtime structures described, the SSCLI also provides access to type metadata, a less efficient repository of *all* class information. The reflection library uses metadata to provide dynamic runtime access to type information.

6.4. The platform adaptation layer

The SSCLI runtime is implemented over the platform adaptation layer (PAL), a virtual subset of the Windows API. This layer provides portability. Much of the work involved in porting the SSCLI to another platform is in implementing the PAL for the target platform**. Because the PAL is essentially a subset of the Windows API, the Windows implementation is thin (for the most part, it remaps PAL functions to their underlying Windows version), whereas the other major implementation, the UNIX PAL, is thicker (the UNIX PAL consists of approximately 4.5 times more lines of code than the Windows PAL). We have not tested whether the PAL implementation has a significant effect on the performance of the SSCLI. However, a difference in performance should be expected.

7. MPICH2

MPICH2 [19] is an implementation of the MPI-1 and MPI-2 specifications, which uses a layered approach for portability (Figure 5). The top layer defines a platform and interconnect generic MPI interface. The next layer is the abstract device interface (ADI), or *device* layer, which defines operations such as message queuing, packetizing, handling heterogeneous communication and data transfer. CH3 is the most common device implementation. CH3 also defines a lower-level *channel* layer, which is specifically responsible for data transfer.

Implementing MPICH2 with a new transport requires developing a new channel [37]. For the simplest port, this requires implementation of five functions, which define the simplest functionality required to move a message from one address space to another [38]. A more comprehensive port might re-implement the entire device.

The default MPICH2 configuration is the sock channel with the CH3 device, providing communication over transmission control protocol sockets. There are two versions of the sock

**To port the SSCLI to another platform, the other major necessary change would be a new JIT implementation for the specific instruction set.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

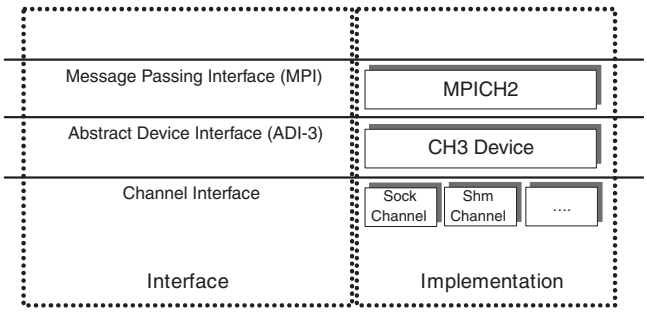


Figure 5. The MPICH2 layered architecture (right) with an example implementation, the CH3 device with the socket channel (left).

channel: Windows and Posix. Other channels include Shm, communication over shared memory; and ssm, communication over shared memory and sockets.

8. MOTOR IMPLEMENTATION

The Motor implementation consists of three parts (Figure 6):

(1) The Microsoft SSCLI, which provides an existing runtime (shown in white).
(2) MPICH2, which provides a portable MPI implementation (shown in gray).
(3) An additional set of new Motor components (shown in black), which implement the OPI libraries and interface to the integrated MPICH2 library.

In addition, Figure 6 illustrates the integration of the layered MPICH2 architecture (shown on the left) within the layered SSCLI runtime environment (shown on the right).

The Motor message passing library has been implemented in a similar manner to the System libraries in the SSCLI and commercial .Net. It consists of a managed library located in the System namespace, which interfaces to the message passing library, within the virtual runtime. The MPC is based on the Windows MPICH2 code-base, ported to the SSCLI PAL, which defines a virtual subset-Windows API. This composition, its individual components and the integration of MPICH are discussed further in the following sections.

System Library: The SSCLI System Library is extended with a managed message passing library, located in the System.MP namespace. This is an interface to the underlying embedded message passing library, including both the MPI and the OPI message passing operations. Sections 8.2 and 8.3 discuss the specific MPI and OPI in detail.

The Motor runtime layer: The SSCLI runtime layer is extended with three major components:

(1) The MPC, an embedded message passing library, which is based on the MPICH2 code-base. This component is the basis of the MPI and the OPI operations. The MPC consists of the parameter checking and collective communication functionality from the MPICH2 MPI interface, and the CH3 code-base, which is an implementation of the ADI-3.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

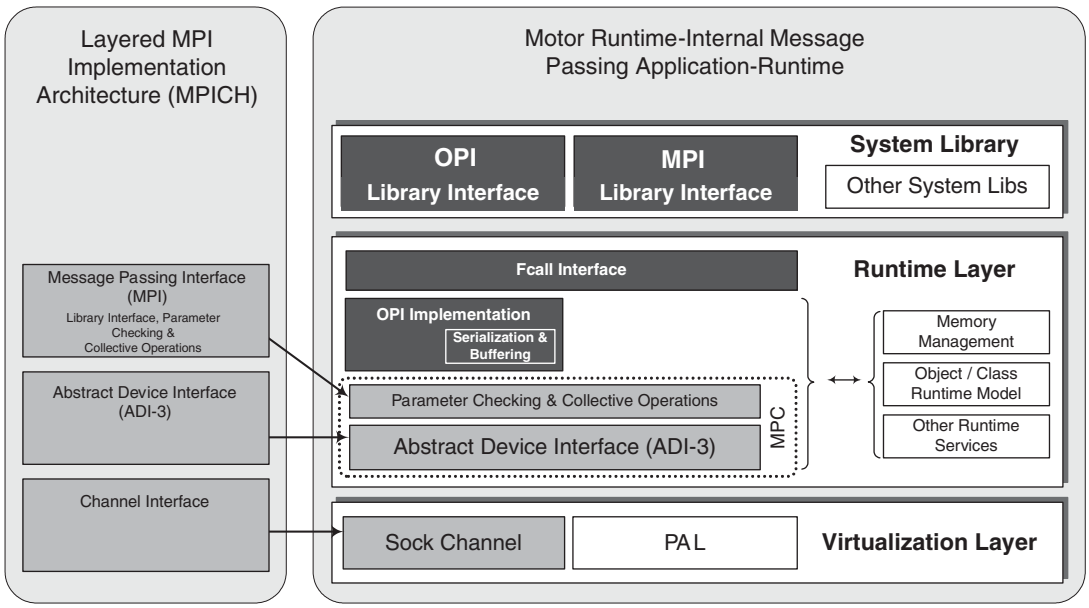


Figure 6. The Motor runtime architecture and its synthesized components. Original SSCLI components are in white. Ported MPICH2 components are in gray. New Motor components are highlighted in black.

(2) An FCall interface that exposes the MPI and OPI operations to the System library. The implementation of this interface is discussed in Section 8.1.2.
(3) The OPI Implementation, a library that maps between the object-oriented library interface and the MPC. This includes structured data transport implementation, which provides serialization and buffering.

Both the MPC and the OPI library interact with the runtime object/class model and managed memory (as illustrated in Figure 6). This interaction is described in further detail in Sections 8.2 and 8.3.

The virtualization layer: The SSCLI PAL has been augmented with a low-level message transport interface, the MPICH Channel Interface. Together these form an overall virtualization layer, over which the Motor Runtime is implemented. The current Motor runtime has been implemented on Windows using the MPICH2 Windows sock channel and the Windows PAL.

8.1. Integration of MPICH2 within the SSCLI

MPICH2 was integrated within the SSCLI in a number of steps:

(1) Porting the MPICH library to the PAL.
(2) Implementing an FCall MPI interface that exposes the embedded message passing library to the System library.
(3) Augmenting MPICH2 to poll for garbage collection for object-oriented languages.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Sections 8.2 and 8.3 further expand on this section by presenting the embedded implementation of the MPI and OPI libraries.

8.1.1. Porting MPICH to the SSCLI PAL

To provide portability, the SSCLI runtime is implemented entirely over the SSCLI virtualization layer, the PAL. Therefore, to integrate MPICH2 within the SSCLI runtime, it also had to be implemented over the PAL.

Because the PAL is a subset of the Windows API, it was decided to use the Windows MPICH2 port [19] as the starting point for this implementation. However, 32 Windows API operations and a much larger number of constants and macros used by the MPICH2 library were unsupported by the PAL. These previously unsupported operations were implemented in two ways:

- Some operations and *all* constants and macros were completely implemented *within* the PAL. For example, simple operations (e.g. htonl, which converts a 32-bit long number to network byte order) were implemented. To implement a new extended version of the PAL for a particular platform, these operations will *not* need to be implemented again.
- The remaining 27 previously unsupported operations were implemented in the PAL and mapped to their underlying Windows implementation (for example, the Windows `GetComputerName` and `GetModuleHandle` operations).

As illustrated in Figure 6, only the lowest layer of MPICH, the channel layer, was not ported to the PAL, because this layer forms part of the Motor virtualization layer.

8.1.2. Exposing the embedded communication library

This section details the manner in which the Motor message passing library is exposed from within the Motor runtime using the SSCLI FCall mechanism. A message passing call will pass through the following three layers, which are illustrated in Figure 7 using an example OPI Recv:

(i) The user code calls the managed MPI or the OPI operation. OPI operations are implemented directly within the System.MP System library, whereas MPI operations are implemented in an external library written in C, which is described in the next section.

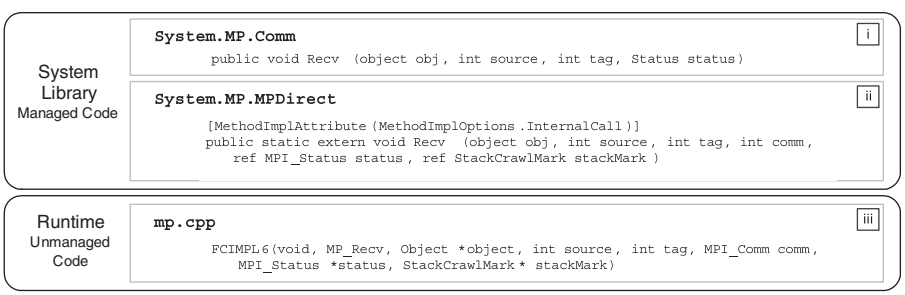


Figure 7. The System library to internal runtime call stack.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



(ii) The System.MP library includes an MPDirect class that contains InternalCall definitions into the underlying runtime. The MPI or the OPI operation calls the matching MPDirect InternalCall function.
(iii) Each MPDirect InternalCall is matched by an embedded FCall.

In the FCall parameters, types are mapped to their embedded C++ equivalent. In the case of the OPI, the provided object is a pointer to the internal representation of a CLI object type: Object. Note that the MP_Recv is defined within a macro (FCIMPL6), which enforces correct parameter ordering.

8.2. Motor MPI implementation

The Motor MPI interface is written in C and compiled to IL using the lcc.NET [39] compiler.

The lcc.NET compiler is an addition to lcc [40], which is a retargetable compiler, meaning that it can be augmented to support a variety of instruction sets and architectures. In the case of lcc.NET, it has been augmented to support the .Net runtime and IL instruction set. The Microsoft C++ Managed compiler was also considered for our work. However, the C++ Managed compiler produces a hybrid managed and native output, which is not supported by the SSCLI.

The Motor MPI library calls the System.MP interface routines into the runtime. However, lcc.NET is an ANSI C compiler and therefore it does not support any object-oriented syntax. Therefore, it does not support any mechanisms to call into the object-oriented System library. For this reason, the Motor MPI interface defines stub MPI functions. A user writes an application in C, or another procedural language, which can be compiled to IL. Once the user application is compiled to IL, these stubs are discarded and the application is re-linked against the operations exposed by the Motor System library.

8.3. Motor OPI implementation

The OPI library is implemented directly within the System.MP library and the object model is based on the official MPI-2 C++ bindings. Figure 3 shows the most significant classes: Comm and Intracomm. Similar to all SSCLI System libraries, the System.MP library is written in C#.

Using access to the runtime object/class model, the MPI implementation checks the provided object attributes and then performs the operations by interfacing to the underlying MPC. For each transport operation, the MPI implementation checks:

- The size of the supplied sending or receiving object, which is used in the underlying data transport.
- Whether the supplied sending or receiving object references other objects, which is not allowed in order to protect object model integrity[††]. If it does, the operation throws an exception.

For OPI operations, the ported MPICH2 library has been augmented to allow an SSCLI `Object*` as the provided buffer. The MPC has been changed to allow sending and receiving data in terms

[††]The SSCLI object/class model keeps track of whether particular object types contain references to other objects, and therefore performing this check is very efficient. This information is kept by the SSCLI to help the garbage collector efficiently perform the mark phase of collection.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

of the internal object representation: `Object`. When required, the MPC resolves the Object to the offset location of its instance data, to pass to the underlying transport.

The following sections describe the interaction between the Motor-embedded message passing library and the SSCLI memory management system. These sections demonstrate the closely coupled interaction between the Motor message passing library and runtime services.

8.3.1. Garbage collection polling

To implement the OPI library, it was essential to ensure that no existing runtime services, such as garbage collection, were disrupted. Because the OPI library uses the FCall interface to call into the runtime, it is essential that the message passing library adheres to the FCall rules. As introduced in Section 6, the SSCLI garbage collector requires all threads to regularly poll the garbage collector. Managed code polling is compiled in by the JIT, whereas FCalls must *explicitly* and *regularly* poll for collection.

Therefore, Motor MPI operations poll at least twice during an operation:

(i) On entry to the FCall, before the operation has commenced.
(ii) Immediately before exiting the FCall, after the operation has been completed.

In addition, blocking operations must poll *during* the operation:

(iii) Threads that enter the message passing library never block when waiting for a transport operation to be completed. Rather they enter a *polling-wait* state to wait for the operation to be completed. To achieve this, blocking system calls in the original MPICH2 were replaced with a *polling-wait*, which periodically releases and polls the garbage collector.

8.3.2. Object pinning

To provide automatic pinning and unpinning of send and receive objects, the Motor MPI implementation applies a *pinning policy*, which decides whether automatic pinning is required for each message passing operation. The basis of this policy is that pinning is not necessary for *every* MPI operation. Rather, it is only required under both of the following conditions:

- If garbage collection might occur.
- If the object has the potential to be moved during that collection.

A pinning policy *cannot* reduce the chances that a garbage collection will occur or that objects will require pinning during that collection. After all, pinning will always be necessary to protect objects that *might* be moved during an impending collection. If a garbage collection occurs then objects that are being accessed by the underlying transport must be pinned. However, a pinning policy does mean that objects are not pinned with *each* MPI operation. Rather, an object is only pinned if it is at risk of being moved and there is a chance of an impending collection. Because MPI is implemented alongside the garbage collector, Motor can determine whether the necessary conditions are satisfied, in which case the object requires pinning. This policy relies on access to the object internal memory location and knowledge of the internal mechanics of the SSCLI two-generational garbage collector (Section 6.2). The pinning policy is different for blocking operations and non-blocking operations.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



8.3.3. Pinning policy for blocking operations

The following describes the Motor pinning policy for blocking operations:

- The sending or receiving object is checked whether it is at risk of being moved during a collection by checking its resident generation.
- If the object exists in the elder generation then it is *not* at risk of being moved during collection. The transport can continue without pinning.
- On the other hand, if the object has been established as a resident of the younger generation, it might be moved during collection.

  Depending on the underlying transport, many operations are completed very quickly. If transport is completed before the thread yields to garbage collection, then pinning is unnecessary. Therefore, pinning can be deferred until the transport operation has been commenced and the thread is about to enter a polling-wait. By the commencement of the polling-wait, the transport operation will either be complete or ongoing:

  - Fast operations can complete almost immediately and thus will be complete as soon as the operation is about to enter the polling-wait. At this point, MPI operations can be completed without pinning because the transport operation has been completed and no longer accesses the object.
  - Ongoing operations need to pin the object, yield to garbage collection and continue waiting for the transport to be completed.

Deferring the decision to pin until the last moment before a possible collection allows quick operations to be completed without pinning.

8.3.4. Pinning policy for non-blocking operations

Non-blocking operations return immediately and therefore they do not provide an opportunity to defer pinning. Furthermore, implementing pinning for non-blocking operations is more problematic because it is not clear when the transport operation has been completed and when the object can be unpinned. Existing external implementations of the MPI, such as the Indiana .NET bindings, do not describe a solution to this problem. The MPJ specification does not detail a solution because, similar to MPI, it defines an MPI and does not stipulate the internal workings. The Motor solution to ensuring that objects are unpinned after transport completion is to augment the garbage collector, which allows for pinning operations that are dependant on the status of an MPI operation. This is described in the following pinning policy for non-blocking operations:

- Objects in the elder generation are not pinned, as in the pinning policy for blocking operations.
- Younger generation objects are pinned immediately before transport is commenced. Then the non-blocking operation returns.

The pinning request is stored by the garbage collector until the next collection. Immediately after the mark phase of collection, when the garbage collector checks for unreferenced memory, the collector iterates through a list of pinning requests and checks the status of non-blocking operations. This causes the collector minimal extra work during this phase of collection. Objects

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

used in *completed* non-blocking operations do not need to be pinned during collection and the object can be released from further pinning. On the other hand, objects used in *incomplete* non-blocking operations are marked as pinned to guarantee that they are not moved during impending collection.

8.3.5. Motor structured data transport implementation

The OPI SDTO perform serialization to a buffer and then interfaces to the underlying MPC. Before sending the serialized buffer, the library performs an MPI send, announcing the size of the serialized buffer[‡‡]. Using this size, the receiver allocates an adequate buffer, receives the serialized buffer and de-serializes it. Pre-announcing the size of the buffer decreases performance; however, it provides the receiver an opportunity to allocate a buffer of adequate size.

Transportable attribute: The Transportable attribute is implemented as a CLI custom attribute, which provides a declarative mechanism to decorate CLI classes. This allows the serialization library to recursively include references that are to be propagated. CLI custom attributes can be introspected using the CLI reflection library. However, the reflection library is relatively slow because it accesses type metadata. Therefore, we decided to augment the Motor class/object model to keep track of transportable attributes. The SSCLI `FieldDesc` structure provides runtime details about each class field. Motor extends the SSCLI `FieldDesc` (Section 6.3) structure to include a transportable bit. This provides an efficient mechanism to determine which object references are transportable. For each object being serialized, the serialization library iterates through all the `FieldDesc` entries for that type. Entries that are Transportable and not null are also serialized.

Serialization and buffering: The Motor serialization mechanism produces a flat object-tree representation with two parts:

- A type table that consists of the fully qualified class names of the transported objects, providing the receiver with information to instantiate the transported objects.
- Object data, which consist of transported objects, laid out side-by-side, each prefixed with an internal-type reference. Serialized objects are copied to the serialization buffer and the object header is replaced with a reference to the specific type in the type table.

The Motor serialization mechanism is recursive and can be described as follows:

(1) The object is copied to the serialization buffer and the object header is replaced with a reference to the fully qualified class name in the type table.
(2) Each field of the object is checked by iterating through the `FieldDesc` entries for that type and checking the transportable bit. Object references are treated differently according to the following rules:
    - A reference to an object that has already been serialized is exchanged to its local serialized reference.
    - A reference to an object that is non-transportable is swapped to null; and
    - A reference to an object that is transportable, but not yet serialized, is recursively serialized.

[‡‡]This technique is also used by mpiJava.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



To support collective communication operations such as scatter, the serialization mechanism can produce a split representation. A single split representation is constructed of many regular representations, each with an individual-type table and each individually deserializable at the receiving end. For scatter operations, the serialization mechanism automatically splits the array and flattens referenced objects. Conversely, for gather operations, the deserialization mechanism takes many split representations and reconstructs them into a single array.

Motor automatically provides buffers for SDTO. These are allocated from static runtime memory, which means that the buffer is not subject to the requirements of pinning. Motor buffers are created on demand and stored in a stack for later use. Each time a buffer is used, it is noted as recently used. The Motor garbage collector has been augmented to check for buffers that have not been recently used and unallocates them.

9. DEMONSTRATION AND PERFORMANCE OF THE MOTOR APPLICATION-RUNTIME

The goal of this section is to the measure the performance of the Motor environment. Section 9.1 discusses the execution, results and analysis of a set of micro-benchmarks, and Section 9.2 presents the implementation and performance of a scientific model, the shallow water equations, developed for the Motor environment.

Throughout these tests, all C# applications were compiled to the CLI IL using Visual Studio 7.1.3088. Similarly, C++ applications were compiled to Win32 native using Visual Studio 7.1.3088 with the speed optimization turned on. Standard C applications were compiled to the CLI IL using lcc.NET. Java applications were compiled using the Sun Java JDK 1.5.0_04 javac compiler and executed using the Sun Java JDK 1.5.0_04 runtime. MPICH-2 v1.0.2 was the common MPI implementation used by *all* applications, apart from Motor applications. However, Motor itself is a product of the same version of MPICH-2. The Indiana C# bindings and mpiJava were all originally implemented over MPICH, which does not perform as well as MPICH-2. To provide a fair comparison, we re-implemented them over MPICH-2 v1.0.2. All applications tested, including both the micro-benchmarks and the shallow water applications, used the MPICH2 socket channel for communication—the only implementation of the channel interface currently implemented within Motor.

9.1. Motor micro-benchmarks

To evaluate the performance of the Motor MPI library we compared a number of applications that implement a Ping-Pong algorithm, where two processes take turns to send and receive a block of data. Our intention is to measure the efficiency of the message passing facilities provided by the Motor environment and purposefully avoid measuring the performance of the environment or data processing. In contrast, Section 9.2 is devoted to presenting the measurements and analyses of the performance of the overall Motor environment.

Both the MPI and OPI operations (Section 9.1.1) and the OPI structured data transport (Section 9.1.2) operations were tested.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

9.1.1. The MPI/OPI micro-benchmark

To test the performance of the MPI/OPI operations, the following implementations of the Ping-Pong code were tested:

(1) A native C++ application using MPICH-2 v1.0.2.
(2) A C application compiled for Motor and using the Motor MPI library.
(3) A C# application compiled for Motor and using the Motor OPI library.
(4) A C# application using the Indiana C# .Net bindings, hosted by the SSCLI.
(5) A C# application using the Indiana C# .Net bindings, hosted by .Net version 1.1.
(6) A Java application using mpiJava version 1.2.5.

For the purposes of this experiment, a single iteration is defined as the round trip between the two participating processes. Each experiment performed 200 iterations, the last 100 of which were timed. This ensured that the overhead of environment startup, communication startup and JIT compilation was avoided in the performance measurement. A range of buffer sizes were tested and each buffer size was tested five times and the average time in microseconds per iteration was calculated. All tests were performed on a Pentium M 1.7 GHz, with 1 GB RAM, running Windows XP SP2. A single node was used because we were interested only in the performance of the message passing implementation, rather than the underlying transport.

Figures 8 and 9 show the performance results for small and large buffer sizes, respectively. In addition, Figure 10 shows the bandwidth measured by the Ping-Pong micro-benchmark. As expected, the native C++ application shows the best performance. Significantly, the two applications executed by Motor show the next best performance. In comparison to the native C++ implementation, the C (Motor) application performs 3% slower across all buffer sizes and 9% slower across buffer sizes less than 8192 bytes. The C# (Motor) application performs 8% slower across all buffer sizes and 17% slower across buffer sizes less than 8192 bytes.

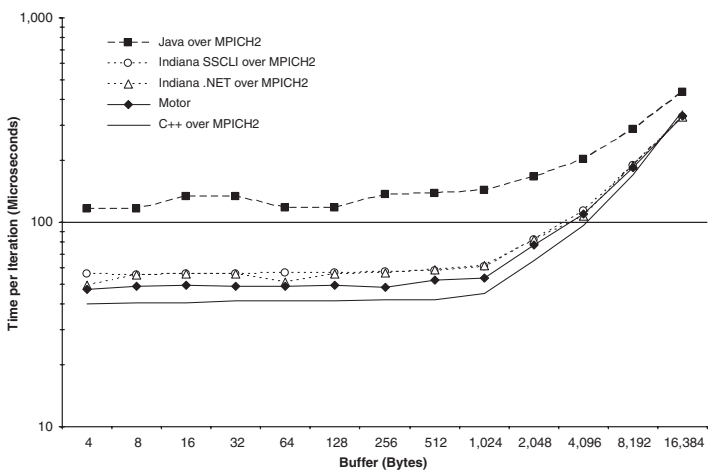


Figure 8. Ping-Pong comparison of MPI operations showing time per iteration (4–16 384 bytes).

Copyright © 2008 John Wiley & Sons, Ltd.

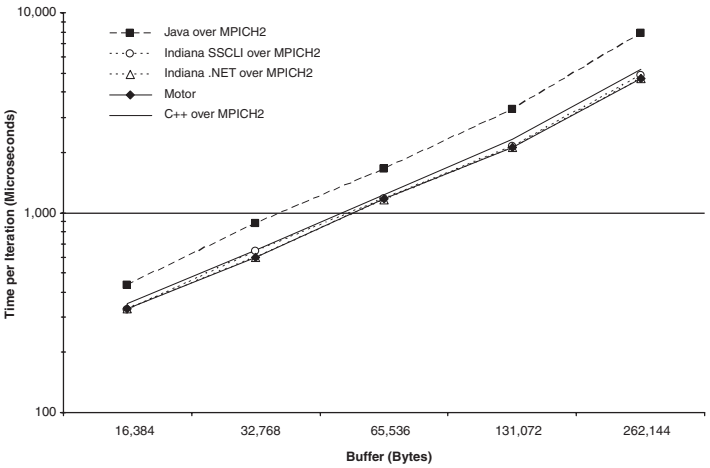


Figure 9. Ping-Pong comparison of MPI operations showing time per iteration (16 384–262 144 bytes).

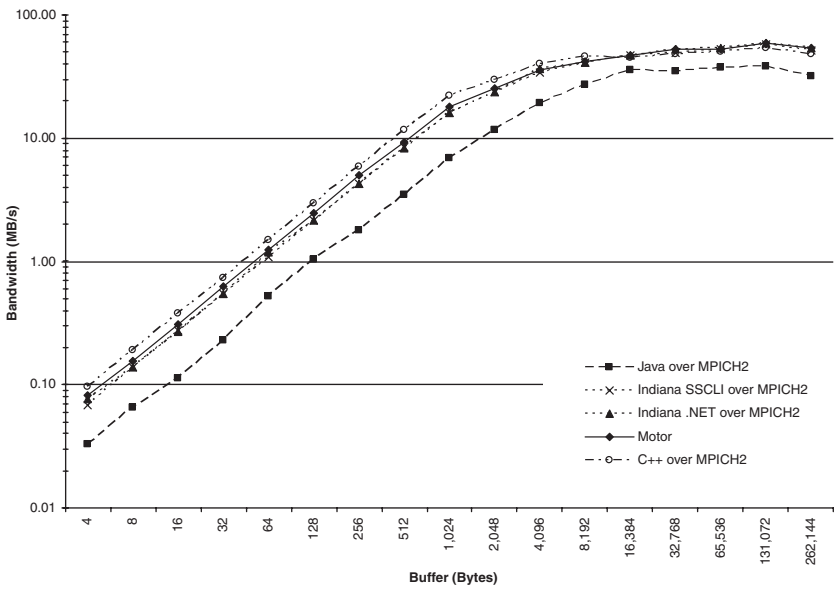


Figure 10. Ping-Pong bandwidth by message size.

In our tests, mpiJava performs the least favorably. We have examined published mpiJava performance results [11]; however, recreating the same results that were published was problematic for a number of reasons. First, the machines used are obviously different. Second, [11] does not specify the exact version of the Java virtual machine employed. Third, [11] indicates that the overhead of

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

mpiJava varies depending on the underlying MPI implementation. As discussed in Section 9, we have re-implemented mpiJava over MPICH-2 because using the original MPICH does not provide a fair comparison—all other environments we tested use the more highly tuned MPICH-2.

In particular, C# (Motor) performs noticeably better than the C# application using the Indiana bindings hosted by the SSCLI using MPICH-2: 16% at a peak, 8% on average over all buffer sizes and 3% on average over buffer sizes greater than 65 536 bytes. These are encouraging results considering that Motor is a synthesis of these same major components: the SSCLI and MPICH-2. We attribute these results to the fact that the Motor MPI library uses the fast InternalCall mechanism, which is a *trusted* internal call mechanism and does not perform data marshaling. Therefore, the InternalCall mechanism imposes a smaller overhead than managed-to-native call mechanisms such as P/Invoke.

9.1.2. The structured data transport micro-benchmark

To test the performance of the Motor OPI SDTO we also used a Ping-Pong code similar to the previous test. However, this experiment measured the influence of sending and receiving structured data, serialized and deserialized using the particular mechanism supported by each system. The following implementations of the Ping-Pong code were compared:

(1) A C# Motor application, using the Motor SDTO;
(2) a Java mpiJava application, using the mpiJava MPI.Object data type, which uses the Java serialization mechanism to transport structured data and
(3) an Indiana C# .Net application, hosted by both the SSCLI and commercial .NET. To provide structured data transport, we used the CLI binary serialization mechanism to produce a buffer to be transported using the standard MPI routines.

Procedural C applications were not included in this test because standard MPI does not define a facility to *automatically* transport structured data. The structured data were in the form of a linked list, with each list element containing a buffer. Figure 11 shows the LinkedArray data structure written in C#, which was used in both the Motor and Indiana applications. The Java application transported an identical data structure.

The total data buffer was 4096 integers, evenly distributed over the entire linked list. The number of linked list elements varied from 1 to 4096[§§]. Each experiment was executed 10 times and the average time in microseconds per iteration was calculated.

Figure 12 plots the results of the structured data transport micro-benchmark. The Motor OPI serialization mechanism performs best throughout the range of linked list lengths (it executes an average of 65% faster than the .Net serialization mechanism). We found the bump in mpiJava results to be a consistent feature and we believe it might suggest that Java version 1.5.0_04 employs different serialization algorithms or data structures to serialize small and large numbers of objects. The mpiJava results stop at 1024 objects because longer-linked lists caused a stack overflow exception in the Java serialization mechanism. It is interesting to note the difference in performance of the .Net and SSCLI serialization mechanisms. It is clear from the results that the serialization mechanism

[§§] The total number of objects transported was twice the number of linked list elements because the array referenced by each linked list element is itself an individual object.

Copyright © 2008 John Wiley & Sons, Ltd.

```
class LinkedArray
{
   public int[] array = null;
   public LinkedArray next = null;
}
```

Figure 11. The LinkedArray class.

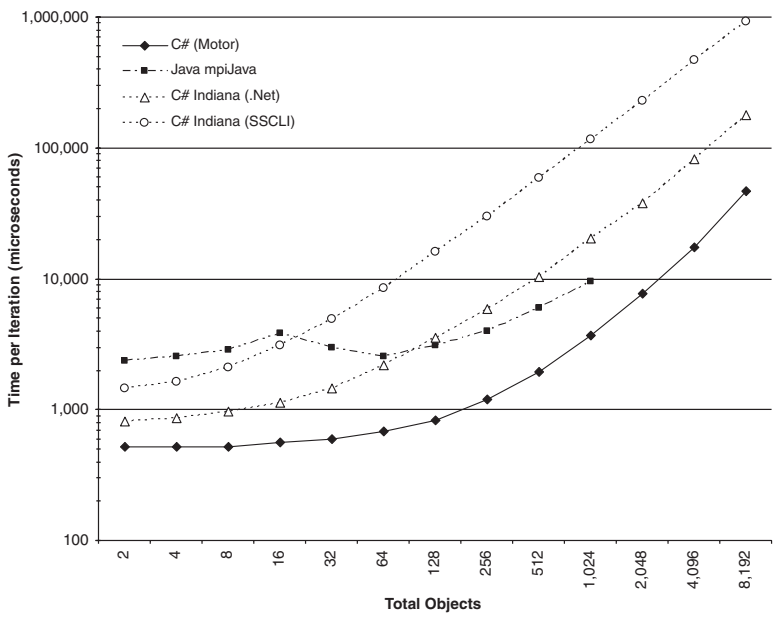


Figure 12. Ping-Pong comparison of structure data transport operations.

employed in the SSCLI is inferior to that provided by .Net. Examining the SSCLI source code, it becomes apparent that the SSCLI uses the reflection library to introspect objects using metadata. We believe it is likely that the .Net serialization mechanism is embedded within the runtime, meaning that it has faster access to object information.

9.2. Shallow water case study

This section demonstrates the application of the Motor runtime and analyzes its suitability for hosting high-performance e-Science applications. To achieve this we have developed a Motor implementation of the shallow water equations [41], which are a computational fluid model commonly used in atmospheric science codes to simulate the horizontal flow of fluid over a number of time steps. Analyzing an implementation of the shallow water model is particularly interesting because it is a relatively simple example of a computationally intensive fluid dynamics model.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

```
struct Res {
   double   row[m];
   int   indx;
};
```

Figure 13. The `Res` structure.

The parallel version of the shallow water model we implemented uses a master–slave design and works as follows. The master initializes a set of multi-dimensional arrays that represent a geometric space and its attributes. It sends a complete copy of these data structures to each slave process, which commence processing a particular slice of the data. Processing is performed in a time-step loop, where for each iteration, each slave performs computations on its data slice, and then synchronizes the edges of its slice with its immediate neighbors. After the time-step loop, the slaves copy their slice of the data back to the master process, which recomposes the original buffer and performs final processing.

Our C and C# implementations of the parallel shallow water model is based on an earlier implementation, which is written in C [42]. This version uses regular MPI_Send and MPI_Recv operations for the majority of communication between slaves. To communicate results back to the server, it defines an MPI custom data type to send and receive a simple data structure of type `Res` (Figure 13), which represents a data block and its position within the overall data set.

The advantage of parallelizing the shallow water model has been documented [42] and this paper does not investigate or demonstrate speedup—rather we compare different implementations of the parallel shallow water model in order to understand Motor performance.

9.2.1. Implementing the shallow water model for the Motor runtime

We implemented two versions of the shallow water model. The first is a C implementation that is essentially identical to the earlier C version and requires no porting.

The second is a C# implementation. Porting the C shallow water implementation to C# using the Motor application-runtime involved a number of steps:

(1) Porting the overall program structure to a C# object-oriented environment. This involved making changes to implement an overall class structure. Simple data types were mapped to their C# equivalent. For example, `char*` was mapped to the C# `string` class.
(2) Porting the shallow water computational processing routines. C# uses C syntax and therefore, changes to the processing code were minimal. For example, C# uses a different syntax to access multidimensional array elements (`[index1,index2]` versus the original C `[index1] [index2]`). Mathematical library calls were mapped to their CLI System library equivalent. Other library calls, such as console output, were similarly mapped.
(3) Porting MPI functionality to the Motor supported MPI library. Regular MPI primitives, such as MPI_Send and MPI_Recv, were mapped to their Motor equivalent, MP.Send and MP.Recv. This involved removing data type and count parameters because they have been removed from the Motor MPI interface. In the case of array subset transfers, the offset and length parameters were used.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



(4) The original C code uses a custom data type to send and receive the `Res` data structure. In the Motor shallow water implementation we used the SDTO to automatically transport `Res` objects.

In total, the C# Motor shallow water model consists of around 1100 lines of code.

9.2.2. Case study execution

In order to analyze the execution of the Motor shallow water model we ran and compared three versions of the shallow water model:

(1) A Windows native port of the native shallow water application. This implementation uses regular MPI buffer-to-buffer send and receive operations for communication between slaves, and an MPI custom data type to transport the `Res` data structure back to the master.
(2) A C implementation, targeting the Motor application-runtime. This implementation uses the same communication primitives as the native port, but interfaces to the Motor MPI library.
(3) A C# implementation, targeting the Motor application-runtime. As introduced in the preceding section, this implementation uses regular object-to-object MPI send and receive operations for communication between slaves. It uses SDTO to automatically transport the `Res` data type back to the master.
(4) A CLI implementation of the shallow water equations, which is written in C# and is identical to the Motor shallow water implementation, apart from the use of the Indiana C# bindings for communication. We executed this version of shallow water model using both the commercial .Net and the SSCLI. This implementation uses the regular MPI send and receive operations for communication between slaves. It uses the standard CLI serialization mechanism, as originally described in Section 9.1.2, to automatically transport the `Res` data type.

During the experiment each application was executed for 100 time steps. The shallow water matrix sizes were varied, from 8×8 double-length floating point numbers to 512×512 double-length floating point numbers. Each application was executed 20 times for each matrix size and the average result was calculated. All tests described in this section were performed on a computer with dual central processing unit Pentium III 700 MHz processors, with 256 MB of RAM, running Windows 2000 SP2. In each case, the application was executed with three processes, the master and two slaves.

In order to profile the execution of these applications in detail, the slave processes were developed to track processing and communication times separately.

9.2.3. Shallow water results

Figure 14 shows the overall shallow water master execution times. On average, the native implementation performs the best, followed by .Net, then Motor and then the SSCLI. The C (Motor) implementation was only executed up to 128×128 double-length floating point numbers because greater buffer sizes caused a stack overflow exception[¶¶].

[¶¶]This is due to the fact that the lcc.NET compiler allocates memory on the stack, and very large buffer sizes exceed the maximum stack size possible on the target machine. Static allocation of large buffers also failed.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

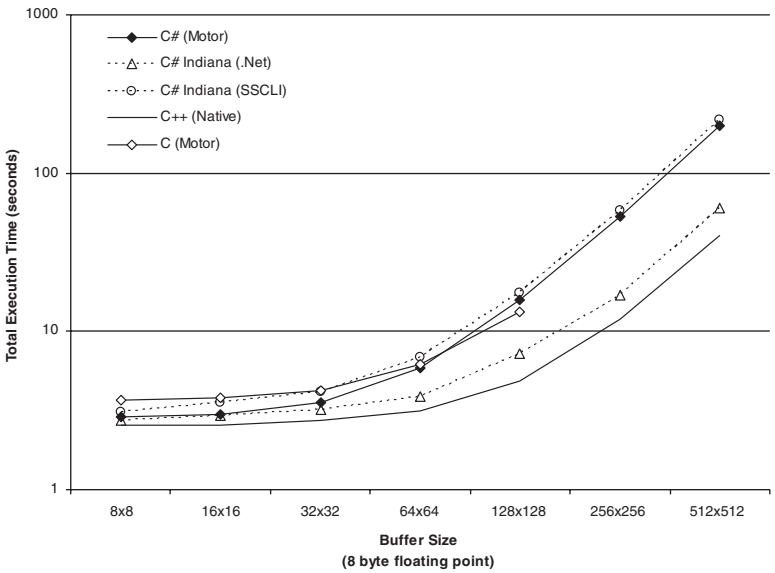


Figure 14. ShallowWater total execution times.

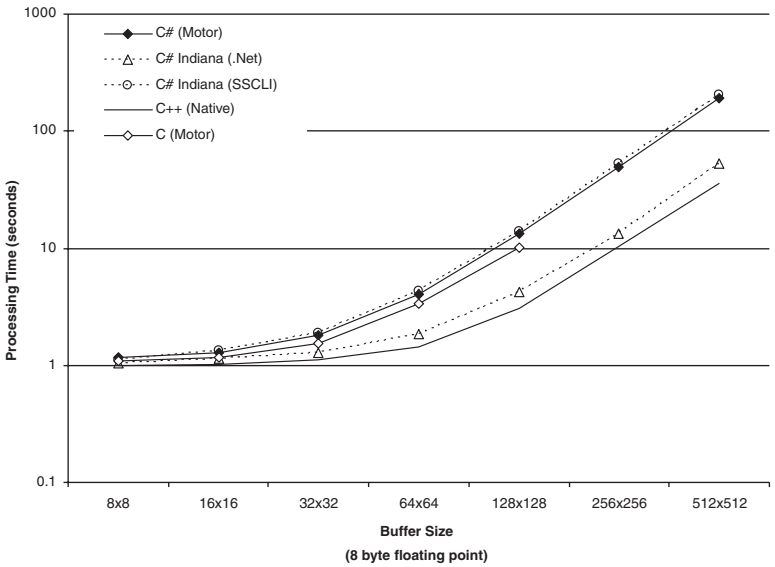


Figure 15. ShallowWater slave processing times.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

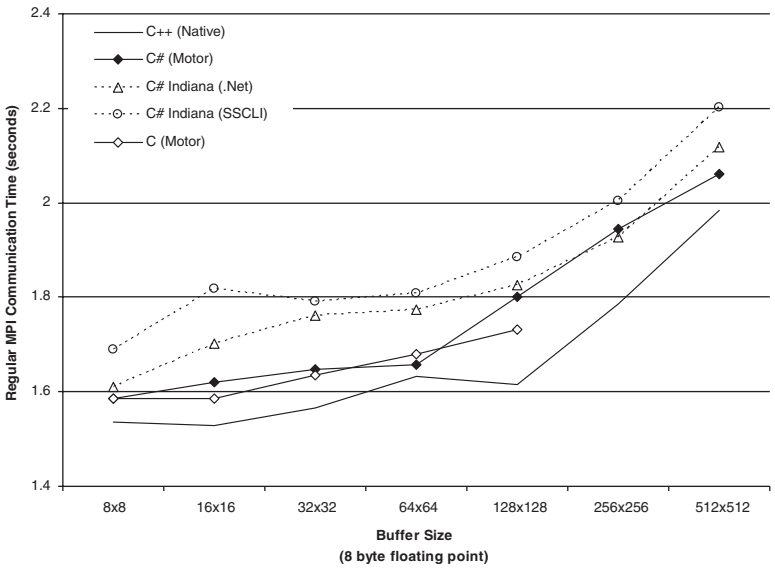


Figure 16. ShallowWater slave regular MPI communication times.

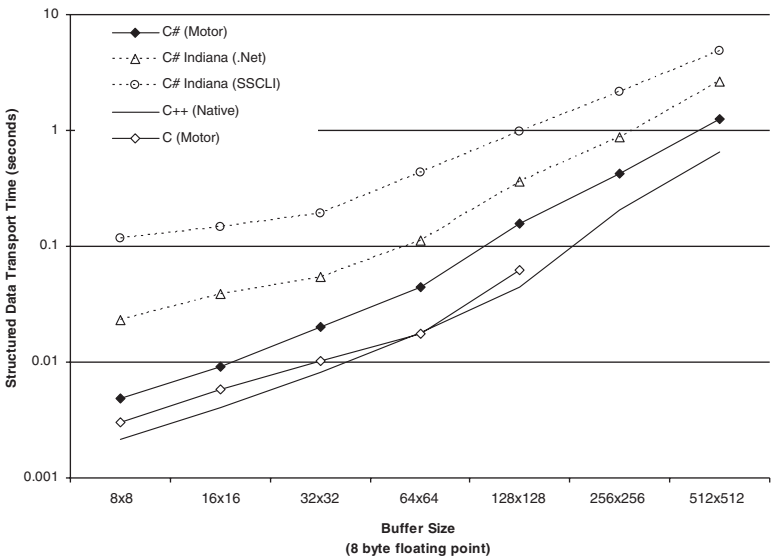


Figure 17. ShallowWater slave-structured data transport times.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

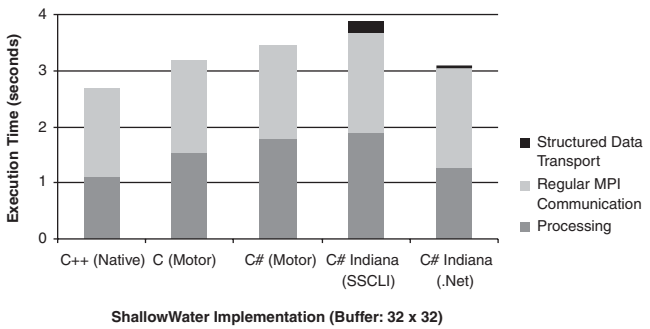


Figure 18. ShallowWater execution profile with a 32×32 buffer size.

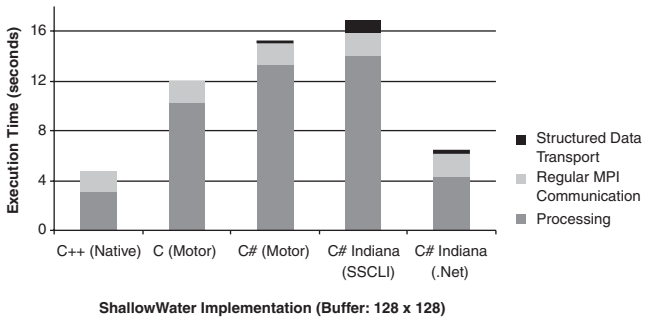


Figure 19. ShallowWater execution profile with 128×128 buffer size.

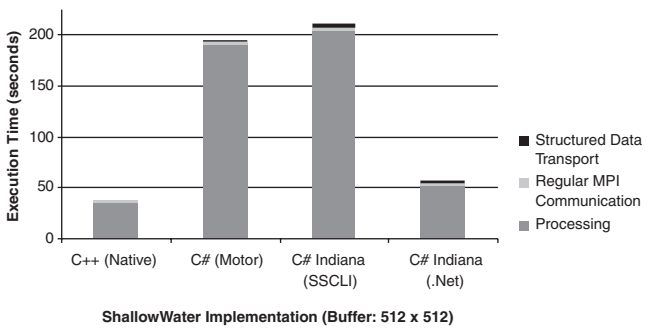


Figure 20. ShallowWater execution profile with 512×512 buffer size.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe



Figures 15–17 decompose the slave execution times into three parts:

(1) Processing time (Figure 15), which includes all executions apart from communication;
(2) regular MPI communication time (Figure 16), which includes regular MPI routines such as MPI_Send and MPI_Recv, or their environment-specific equivalent; and
(3) structured data transport time (Figure 17), which encompasses the time required to transport the `Res` data structure, using the environment-specific mechanism.

In addition, Figures 18, 19 and 20 illustrate the overall communication and processing profile of each shallow water implementation, at buffer sizes 32×32, 128×128 and 512×512, respectively.

Figures 18–20 show that at large buffer sizes, the shallow water applications profiled are bound by processing. Furthermore, Figures 15 and 20 illustrate that processing time is the cause of the poorer performance demonstrated by both Motor and the SSCLI. Because Motor is based on the SSCLI, it uses the same JIT compiler and therefore, it is logical that they demonstrate similar processing performance.

On average across all measured buffer sizes, our Motor prototype performs significantly worse than the native implementation (the C implementation performs 83% slower, whereas the C# implementation performs 159% slower). This is because the SSCLI lacks a competitive JIT compiler [16]. Our processing performance results are consistent with published benchmarking results of various implementations of the CLI. These benchmarks illustrated a large difference in the performance of various JIT compilers and found that the foremost factor in runtime processing performance is the optimization of the JIT-generated code [16]. This benchmarking study also found that the SSCLI emitted poorer JIT-generated code in comparison with .Net and Mono. In comparison, .Net, which *does* support an optimized JIT compiler, achieves significantly better results—on average it performs only 28% worse than the native implementation. We believe that the .Net performance is a reasonable tradeoff considering the range of services that .Net provides, including platform independence, language neutrality, memory management and an object model. Although the current Motor processing time is not as good as that of .Net, we are confident that, considering that the SSCLI is a downgraded version of .Net, a more competitive SSCLI JIT implementation could achieve equal results. It is clear that the SSCLI JIT compiler is substandard [16] and optimizations [43,44] could be applied.

With regard to regular MPI communication, Figure 16 shows that both applications hosted by Motor perform favorably against both .Net and the SSCLI. On average, C (Motor) performs 21.9% faster than C# (SSCLI) and 9.7% faster than C# (.Net). On average, C# (Motor) performs 6.9% faster than C# (SSCLI) and 3.7% faster than C# (.Net). On average, C (Motor) performs only 4.5% slower than the native implementation. These MPI communication times are consistent with our Ping-Pong micro-benchmarks.

Similarly, with regard to structured data transport of the `Res` type, Figure 17 demonstrates that Motor performs significantly better than .Net and the SSCLI, whereas the native implementation performs best. The native and C (Motor) shallow water implementations transport the `Res` type most efficiently because they both use a custom data type, which does not involve any serialization; C (Motor) performs 30% slower than the native implementation.

The Motor OPI SDTO are significantly faster than the regular serialization mechanisms offered by SSCLI and .Net. On average C# (Motor) performs nearly 8 times faster than the C# (SSCLI) and 2.6 times faster than C# (.Net).

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

In C/C++, using an MPI custom data type is possible because the C/C++ definition of the `Res` type is a single discreet structure and therefore the `Res` type can be transported with a single buffer-to-buffer transfer. However, in C# and Java, the `Res` object contains a reference to separate the array object and therefore a single object-to-object transfer is impossible. This demonstrates the fact that in a managed memory environment, the developer has less control over memory and it is difficult to organize memory to optimize communication. Therefore, an efficient serialization mechanism is essential to transport structured data.

From these results we can conclude that Motor communication performs favorably but its JIT compiler is poor. The performance of the .Net runtime demonstrates that an optimized JIT compiler is able to achieve reasonable results in comparison with a native C++ implementation. To make Motor a competitive environment would require implementing a more competitive JIT compiler or tuning the existing SSCLI compiler.

10. CONCLUSION

Motor is a unique environment for high-performance application development because it supports high-performance communication as part of its overall architecture. Motor places message passing alongside other CLI runtime libraries and services. This provides a guaranteed environment for HPC. It also gives the message passing library access to the entire internal runtime allowing us to implement an efficient object passing interface while guaranteeing object model integrity. Notably, because Motor is based on the CLI, it is language neutral, and it supports message passing libraries for both procedural and object-oriented languages.

We have demonstrated Motor performance using a micro-benchmark measuring regular buffer-to-buffer MPI and our own OPI object-to-object and structured data operations. In addition, we have implemented a scientific model for the Motor runtime and analyzed its performance. This has demonstrated that an embedded message passing architecture is feasible and offers good performance, in comparison with implementations that place message passing outside the runtime environment. We have also demonstrated the importance of a highly optimized JIT compiler, which is not supported by the SSCLI or Motor. However, examining .Net processing performance demonstrates that an optimized JIT compiler is able to provide good processing performance. A post-prototype implementation of Motor would need to support a more optimized JIT compiler.

The Motor architecture introduces and demonstrates the feasibility of developing an overall runtime for HPC. Although our starting point supports message passing, other services, such as shared memory, process management and mathematics libraries, are also important to parallel developers. For example, we believe that the detailed design of an embedded shared memory architecture would also be of significant value. This is particularly true considering the commoditization of multi-core microprocessors. Environments such as OpenMP are becoming more important for HPC developers—and all developers in general—who wish to take advantage of the thread-level parallelism offered by a multi-core architecture.

REFERENCES

1. *Standard ECMA-335*: *Common Language Infrastructure* (3rd edn). June 2005. http://www.ecma-international.org/publications/standards/Ecma-335.htm [11 January 2006].

Copyright © 2008 John Wiley & Sons, Ltd.



2. Foster I, Kesselman C. *The Grid*: *Blueprint for a New Computing Infrastructure* (1st edn). Morgan Kaufmann: Los Altos, CA, 1998.
3. Goscinski W, Abramson D. Application deployment over heterogeneous Grids using distributed ant. *First International Conference on e-Science and Grid Technologies* (*e-Science 2005*), Melbourne, Australia, 2005; 361–368.
4. Goscinski W, Abramson D. Motor: A virtual machine for high performance computing. *The Fifteenth IEEE International Symposium on High-Performance Distributed Computing* (*HPDC2006*), Paris, France, 2006.
5. Goscinski W, Abramson D. An infrastructure for the deployment of e-Science applications. In *High Performance Computing and Grids in Action*, Grandinetti L (ed.). *International Symposium on High Performance Distributed Computing 2007 (HPDC2007)*, Advances in Parallel Computing, vol. 16. IOS Press: Amsterdam, 2008; 131–149.
6. Abramson D. Applications development for the computational Grid (*Lecture Notes in Computer Science*, vol. 3841). Springer: Berlin, 2006.
7. Judd G, Clement M, Snell Q, Getov V. Design issues for efficient implementation of MPI in Java. *Proceedings of the ACM 1999 Java Grande Conference*, San Francisco, U.S.A., 1999.
8. Lee H-K, Carpenter B, Fox G, Lim SB. HPJava: Programming support for high-performance Grid-enabled applications. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, U.S.A., 2003.
9. Thurman D. JavaPVM. http://www.isye.gatech.edu/chmsr/JavaPVM/ [12 January 2000].
10. Morin S, Koren I, Krishna C. JMPI: Implementing the message passing standard in Java. *International Parallel and Distributed Processing Symposium*: *IPDPS 2002 Workshops*, Fort Lauderdale, U.S.A., 2002.
11. Baker M, Carpenter B, Fox G, Ko SH, Lim S. mpiJava: An object-oriented Java interface to MPI. *International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, 1999.
12. Carpenter B, Getov V, Judd G, Skjellum A, Fox G. MPJ: MPI-like message passing for Java. *Concurrency and Computation: Practice and Experience* 2000; **12**(11):1019–1038.
13. Bull J, Westhead M, Kambites M, Obdrzalek J. Towards OpenMP for Java. *Proceedings of Second European Workshop on OpenMP*, Edinburgh, 2000.
14. Dincer K. Ubiquitous message passing interface implementation in Java: jmpi. *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico, 1999.
15. Mintchev S. Writing programs in JavaMPI. *Technical Report MAN-CSPE-02*, 1997.
16. Vogels W. HPC.NET—are CLI-based virtual machines suitable for high performance computing? *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, Phoenix, U.S.A., 2003.
17. Willcock J, Lumsdaine A, Robison A. Using MPI with C# and the common language infrastructure. *Concurrency and Computation: Practice and Experience* 2005; **17**(7–8):895–917.
18. The Java Grande Forum, 2003. http://www.javagrande.org/ [11 April 2007].
19. MPICH2, 2006. http://www-unix.mcs.anl.gov/mpi/mpich2/ [12 September 2006].
20. Shared Source Common Language Infrastructure 1.0 Release, 2003. http://msdn.microsoft.com/net/sscli/ [13 January 2006].
21. Stutz D, Neward T, Shilling G. *Shared Source CLI Essentials* (1st edn). O'Reilly: Sebastopol, CA, Farnham, 2003.
22. Adabala S, Chadha V, Chawla P, Figueiredo RJO, Fortes JAB, Krsul I, Matsunaga AM, Tsugawa MO, Zhang J, Zhao M, Zhu L, Zhu X. From virtualized resources to virtual computing grids: The In-VIGO system. *Future Generation Computer Systems* 2005; **21**(6):896–909.
23. Zhao X, Borders K, Prakash A. SVGrid: A secure virtual environment for untrusted Grid applications. *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*, Grenoble, France, 2005.
24. Krsul I, Ganguly A, Zhang J, Fortes JAB, Figueiredo RJ. VMPlants: Providing and managing virtual machine execution environments for Grid computing. *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, Pittsburgh, 2004.
25. Keahey K, Foster I, Freeman T, Zhang X, Galron D. Virtual workspaces in the Grid. *Eleventh International Euro-Par Conference*, Lisbon, Portugal, 2005.
26. Zhang X, Keahey K, Foster I, Freeman T. Virtual cluster workspaces for Grid applications. *ANL Technical Report*, *ANL/MCS-P1246-0405*, 2005.
27. Foster I, Freeman T, Keahey K, Scheftner D, Sotomayor B, Zhang X. Virtual clusters for Grid communities. *IEEE International Symposium on Cluster Computing and the Grid*, Singapore, 2006.
28. Figueiredo R, Dinda P, Fortes J. A case for Grid computing on virtual machines. *Proceedings of the 23rd International Conference on Distributed Computing Systems*, Washington, U.S.A., 2003.
29. Ganguly A, Agrawal A, Boykin P, Figueiredo R. WOW: Self-organizing wide area overlay networks of virtual workstations. *Proceedings of the Fifteenth International Symposium on High Performance Distributed Computing* (*HPDC-15*), Paris, France, 2006.
30. Castain R, Woodall T, Daniel D, Squyres J, Barrett B, Fagg G. The open run-time environment (OpenRTE): A transparent multi-cluster environment for high-performance computing. *Euro PVM/MPI 2005*, Sorrento, Italy, 2005.

Copyright © 2008 John Wiley & Sons, Ltd.

31. Migliardi M, Sunderam V, Geist A, Dongarra J. Dynamic reconfiguration and virtual machine management in the Harness metacomputing system (*Lecture Notes in Computer Science*, vol. 1505). Springer: Berlin, 1998.
32. Baker M, Carpenter B. Thoughts on the structure of an MPJ reference implementation. *Technical Report CRPC-TR99803*, 1999.
33. Java Native Interface, 2007. http://java.sun.com/j2se/1.4.2/docs/guide/jni/.
34. Clark J. Calling Win32 DLLs in C# with P/Invoke. *MSDN Magazine*, 2003. http://msdn.microsoft.com/msdnmag/issues/03/07/NET/.
35. Parrot Virtual Machine, 2006. http://www.parrotcode.org/.
36. Jones R, Lins R. *Garbage Collection*, *Algorithms for Automatic Dynamic Memory Management*. Wiley: New York, 1997.
37. Gropp W, Lusk E. Creating a new MPICH device using the channel interface. *Technical Report ANL/MCS-TM-213*, 1995.
38. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 1996; **22**:789–828. Available at: http://citeseer.ist.psu.edu/gropp96highperformance.html [20 September 2006].
39. Hanson DR. lcc.NET: Targeting the .NET common intermediate language from standard C. *Software—Practice and Experience* 2004; **34**:265–286.
40. Fraser CW, Hanson DR. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 1991.
41. Vreugdenhil C. *Numerical Methods for Shallow-Water Flow*. Kluwer Academic Publishers: Dordrecht, 1994.
42. Abramson D, Dix M, Whiting P. A study of the shallow water equations on various parallel architectures. *Fourteenth Australian Computer Science Conference*, Sydney, Australia, 1991.
43. Ishizaki K, Kawahito M, Yasue T, Takeuchi M, Ogasawara T, Suganuma T, Onodera T, Komatsu H, Nakatani T. Design, implementation, and evaluation of optimizations in a just-in-time compiler. *Proceedings of the ACM SIGPLAN Java Grande Conference*, 1999.
44. Suganuma T, Yasue T, Kawahito M, Komatsu H, Nakatani T. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2005; **27**(4):732–785.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Chapter 15: Relevant papers printed in past 12 months

A number of papers have appeared in the last 12 months, and are thus not formally included in this D.Sc. thesis. I have indicated which chapters the papers would have appeared had they been included.

(1)	Schmidberger, J., Bate, M., Reboul, C., Androulakis, S., Phan, J., Whisstock, J., Goscinski, W., Abramson, D. and Buckle, A. "MrGrid: A Portable Grid Based Molecular Replacement Pipeline", PLoS ONE, April 2010, Volume 5, Issue 4.	8
(2)	Abramson, D., Bethwaite, B., Enticott, C. Garic, S., Peachey, T., Michailova, A, and Amirriazi, S "Embedding optimization in computational science workflows", Journal of Computational Science 1 (2010) 41–47.	4, 6
(3)	Abramson, D., Dinh, M., Kurniawan, D., Moench, B. and DeRose, L. "Data Centric Highly Parallel Debugging", IEEE High Performance Distributed Computing (HPDC) 2010, Chicago, June 21st – 25th, 2010.	10
(4)	Bethwaite, B., Abramson, D., Bohnert, F., Garic, S., Enticott, C., Peachey, T., "Mixing the Grid and Clouds: High-throughput Science using the Nimrod Tool Family", "Cloud Computing: Principles, Systems and Applications", Eds Antonopoulos and Gillam, Springer, pp 219-237.	9
(5)	Russel, A.B.M., Abramson, D., Bethwait, B. , Dinh, M., Enticott, C., Firth, S., Garic, S., Harper, I., Lackmann, M., Nguyen, H, Ramdas, T., Schek, S., Vail, M., "An Abstract Virtual Instrument System for High Throughput Automatic Microscopy", ICCS 2010, Amsterdam, 31st May – 2nd June, 2010.	6
(6)	Riley, M J W, Peachey, T., Abramson, D., Jenkins, K W. "Multi-objective engineering shape optimization using differential evolution interfaced to the Nimrod/O tool", 9th World Congress on Computational Mechanics and 4th Asian Pacific Congress on Computational Mechanics in 2010, Sydney, 19-23 July, 2010. http://dx.doi.org/10.1088/1757-899X/10/1/012189	4
(7)	Sher, A., Cooling, M., Bethwaite, B., Tan, J., Peachey, T., Enticott, C., Garic, S., Gavaghan, D., Noble, D., Abramson, A., and Crampin, E. "Global Sensitivity Tool For Cardiac Models: Ionic Currents Balance and Hypertrophic IP3 Transients in Atrial and Ventricular Cell Models", to appear, 32nd Annual International IEEE EMBS Conference of the IEEE Engineering in Medicine and Biology Society in Buenos Aires Sheraton Hotel, Buenos Aires, Argentina during August 31 - September 4, 2010.	4
(8)	Tan, J., Abramson, D. and Enticott, C. "Firewall Traversal in the Grid Architecture", 12th IEEE International Conference on High Performance Computing and Communications (HPCC-2010), Melbourne, Australia, September 1-3, 2010.	13
(9)	Ho, T and Abramson, D. "An Active Data Model", in High Performance & Large Scale Computing, W. Gentzsch, L. Grandinetti, G. Joubert (Editors), IOS Press, Amsterdam, The Netherlands, Volume 18, Advances in Parallel Computing, ISBN 978-1-60750-073-5, 2009. pp 399-421.	11
(10)	Abramson, D, Bethwaite, B, Dinh, M, Enticott, C, Firth, S, Garic, S, Harper, I, Lackmann, M, Nguyen, H, Ramdas, T Russel, A.B.M, Schek, S and Vail, M., "Virtual Microscopy and Analysis using Scientific Workflows", IEEE e-Science 2009, Dec 9 – 11th, Oxford, UK.	6
(11)	Abramson, D., Bethwaite, B., Enticott, C. Garic, S., Peachey, T., Michailova, A, Amirriazi, S and Chitters, R., "Robust Workflows for Science and Engineering", ACM MTAGS09, 2nd Workshop on Many-Task Computing on Grids and Supercomputers, co-located with ACM/IEEE SC09 (International Conference for High Performance, Networking, Storage and Analysis), Portland, Oregon - November 16th, 2009.	6

The following papers are yet to appear

(12) Abramson, D., Bernabeu, M., Bethwaite, B., Burrage, K., Corrias, A., Enticott, C., Garic, S., Gavaghan, D., Peachey, T., Pitt-Francis, J., Pueyo, E., Rodriguez, B., Sher, A. and Tan, J. "High Throughput Cardiac Science on the Grid", to appear, Special issue of Philosophical Transactions of the Royal Society. 8

(13) Kurniawan, D. and Abramson, D. "ISENGARD: an Infrastructure for Supporting e-Science and Grid Application Development", to appear, Concurrency Practice and Experience. 12

(14) Abramson, D., Bethwaite, B., Enticott, C., Garic, S. and Peachey, T. "Parameter Exploration in Science and Engineering using Many-Task Computing", to appear, Special issue of IEEE Transactions on Parallel and Distributed Systems on Many-Task Computing. 6