

Relative Debugging and its Application to the Development of Large Numerical Models

David Abramson † Ian Foster ‡ John Michalakes ‡ R. Sosič †

† School of Computing and Information Technology
Griffith University
Brisbane, QLD 4111
Australia
{D.Abramson,R.Sosic}@cit.gu.edu.au

‡ Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
U.S.A.
{foster,michalak}@mcs.anl.gov

Abstract

Because large scientific codes are rarely static objects, developers are often faced with the tedious task of accounting for discrepancies between new and old versions. In this paper, we describe a new technique called *relative debugging* that addresses this problem by automating the process of comparing a modified code against a correct reference code. We examine the utility of the relative debugging technique by applying a relative debugger called Guard to a range of debugging problems in a large atmospheric circulation model. Our experience confirms the effectiveness of the approach. Using Guard, we are able to validate a new sequential version of the atmospheric model, and to identify the source of a significant discrepancy in a parallel version in a short period of time.

1 Introduction

Large scientific codes are constantly evolving. Refinements in understanding of 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 which allow a user to investigate the execution state of an application program, by for example examining the state of program variables [11, 12, 13, 17]. Significant recent extensions include graphical user interfaces to improve the ease of use, data visualization facilities to aid the interpretation of large and complex data structures, the concept of process groups to aid the management of many independent threads in parallel machines, and support for parallel and distributed debugging [2, 8, 9, 11, 14].

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. Instead, they want to *compare* their execution with the execution of an old, reference program which 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 which 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 [6]. This approach also has its limitations: it can require 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 paper we describe a new debugging paradigm called *relative debugging* 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 paper describes a particular instantiation of this paradigm—the relative debugger Guard—and present 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.

2 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. Hence, the relative debugging process proceeds as follows. The user first formulates a set of assertions about key data structures in 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 paradigm. We introduce them briefly here, and describe them

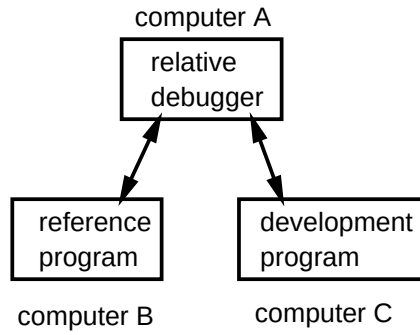


Figure 1: Relative Debugging

in more detail in the next section when we discuss 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 (Figure 1). It is useful for the relative debugger to support the debugging of programs written in different programming languages and executing on different computers in a heterogeneous network. This makes it possible to use the relative debugger when porting programs from one language or computer to another.

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.

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, then the numeric values of differences are printed out. If differences are numerous, then visualization techniques are required to present them in a meaningful way.

3 The Guard Relative Debugger

The Guard relative debugger 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, comes from its specialized commands that support relative debugging. These commands can broadly be broken into two groups, *process control* and *data comparison*.

3.1 Process Control

Guard commands use symbolic names to refer to processes. Names are assigned when Guard launches a new process or when it attaches to an existing one. Guard can handle an arbitrary number of processes concurrently, limited only by operating system restrictions.

Processes are launched using the `invoke` command, which has the following syntax:

```
invoke process_name command [machine_name] [user_name]
```

The `process_name` is a Guard variable which is then used to identify the process. 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 process names as its argument. Execution continues until the processes terminate or encounter a breakpoint or an assertion.

3.2 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 <process_1>::<variable_1> <process_2>::<variable_2>
```

This command traverses two named data structures (**variable_1** and **variable_2**) from the two named processes 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 processes 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 in 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 comparison of data structures.

Assertions are specified by the **assert** command, which has the following syntax.

```
assert [/eps type value] [/file filename]
      <process_1>::<variable_1>@<source_file_1>:line_1 =
      <process_2>::<variable_2>@<source_file_2>:line_2
```

The command contains two tuples, each with a process identifier, a variable name, a source file name and a line number. Guard plants breakpoints in the two processes 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 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, compare, etc. 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 file, which can be overridden using the **/eps** and **/file** parameters in a particular assertion.

3.2.1 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.

3.2.2 Visualizing Differences

Guard supports three approaches to the reporting of differences: text, bitmaps, and advanced visualization techniques. 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 patterns on the display. Arrays with more than two indices can be folded onto two dimensions using a number of standard techniques.

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.

3.3 Underlying Guard Technology

Space does not permit a detailed description of Guard implementation. However, we provide a few relevant details. Guard is implemented above a platform, called Dynascope, which provides an interface for building debuggers [16]. 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 [15]. 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 without modification to any computing platform supported by Dynascope. (Currently, debugging servers have been implemented for Sun, Silicon Graphics, IBM RS6000, and Next computers. Implementations on additional architectures are in progress.) 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 that were being compared running on a Sun computer in Europe and a Next computer in the U.S.A.

4 MM5: A Mesoscale Weather Model

In this section we describe the mesoscale model, called MM5, used in our experiments. MM5 was originally developed at the National Center for Atmospheric Research and has since been modified and ported to parallel platforms at Argonne National Laboratory. The parallel version of the code, called MPMM, is written using a portable parallel library. Whilst it is similar to MM5 in functionality, much of its structure has been altered to allow parallel computations to be performed. Further, some of the key data structures have been altered to allow for data decomposition on distributed memory machines.

4.1 Structure of MM5

The Penn State/NCAR Mesoscale Model is a limited-area primitive equation model, designed to simulate meso-alpha scale (200–2000 km) and meso-beta scale (20–200 km) atmospheric circulation systems [1, 7]. The model may be run at resolutions as low as 1 km. Domains are uniform rectangular grids representing three-dimensional regions of the atmosphere. The horizontal coordinate system is equally spaced geographically and the model uses the Arakawa-B gridding scheme. The vertical coordinate system is σ surfaces, with layers distributed more closely nearer the surface (23 layers in the current model). We use a version with nonhydrostatic atmospheric dynamics [4] and a fourth-order finite difference scheme [3] with a split semi-implicit scheme used to resolve acoustic waves. Physics includes the Blackadar high-resolution planetary boundary layer scheme, the Grell cumulus scheme, explicit moisture with treatment of mixed-phase processes (ice), shallow convection, dry convective adjustment, and the Dudhia long- and short-wave radiation scheme [7].

MM5 is a traditional vector code, written almost entirely in standard Fortran 77. (However, pointers are used to keep track of the different versions of model data structures that correspond to different nesting levels.) Logically, most of the code can be thought of as invoking “physics” routines on a set of independent vertical columns; nevertheless, for maximum vectorization, the code is structured so that the longitude loop is innermost.

4.2 Structure of MPMM

MPMM is a version of MM5 designed to exploit scalable parallel computers [5, 10]. It is intended to be functionally identical to MM5 (that is, it should yield the same results), but is structured quite differently in order to execute efficiently on parallel computers.

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 (Figure 2). It also results in better cache utilization.

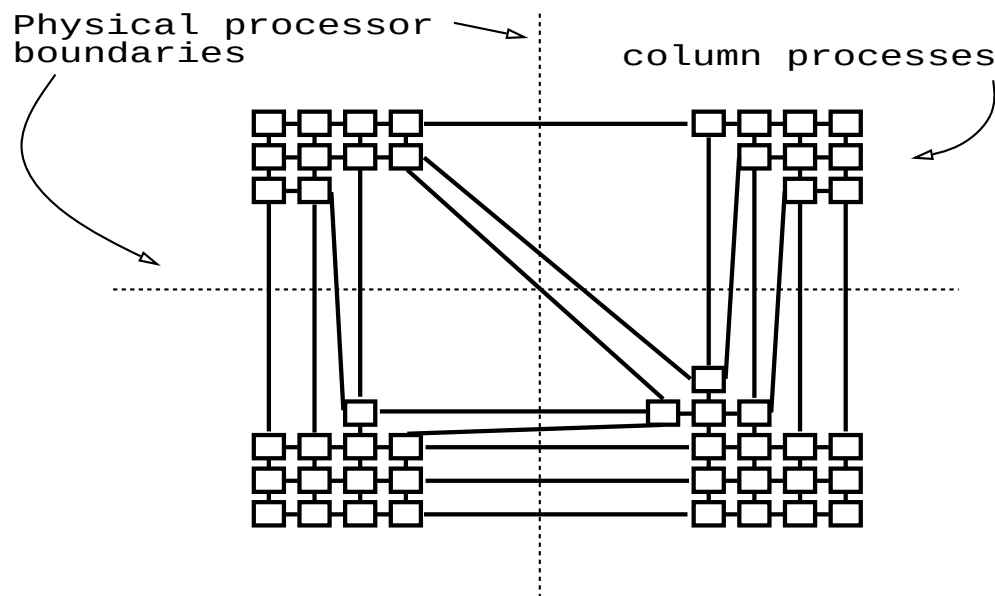


Figure 2: Load balancing in MPMM. This figure shows the situation after columns have been migrated to compensate for load imbalances. Lines indicate data dependencies between columns.

Figure 3 illustrates MPMM code structure. Notice the calls to the RSL routines used to perform computation, to exchange data as required by finite different stencils, and to communicate with nested subdomains. The resulting code performs well on scalable parallel computers, achieving for example 1.15 GFlops on a 64-processor IBM SP2 with Power 1 processors, on a 91 by 91 by 23 domain.

In summary, MM5 and MPMM are two complex numerical codes which, while functionally equivalent, have very different structures. The top-level code structure, loop nesting, and array structures are all quite different. These differences render the comparison of the two models a nontrivial exercise.

5 Debugging Experiments

In this section we describe the experiments performed using Guard and MM5. 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. This task was particularly challenging because of the different data and code structures.

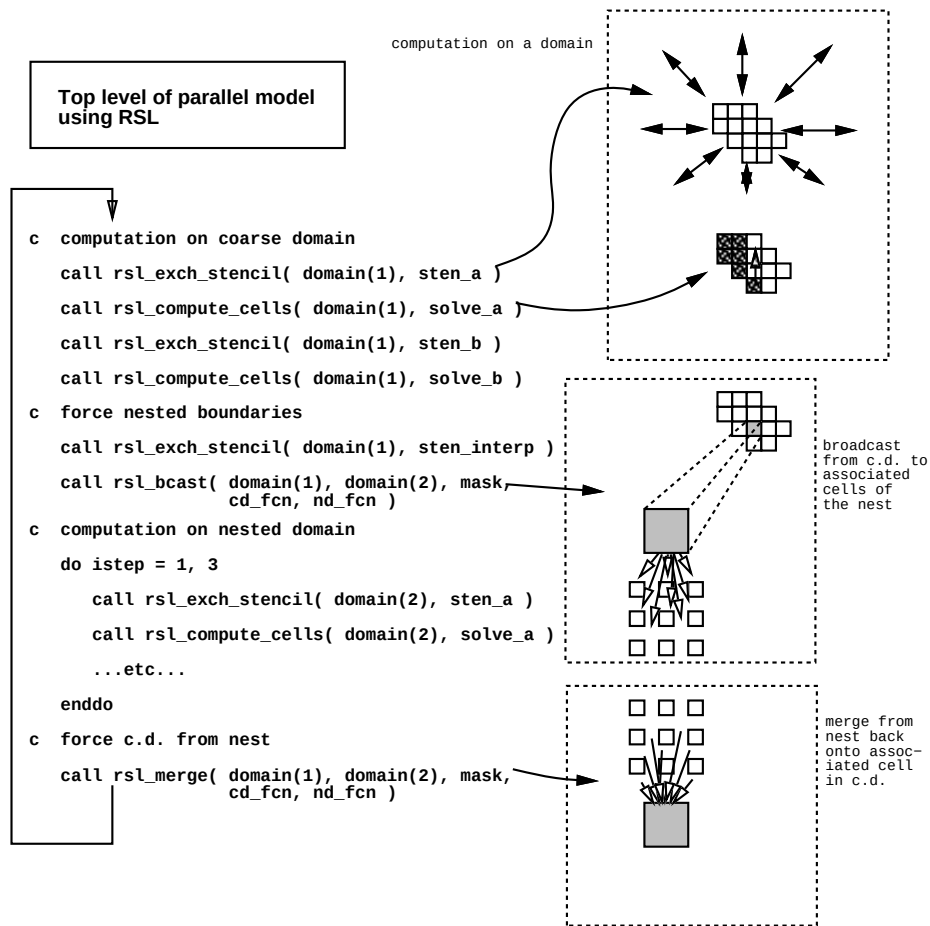


Figure 3: Top level parallel driver for an MM5 time step with nest interactions

5.1 Comparing Sequential Codes

In our first attempt to use Guard to identify previously unsolved problems, we compared 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 that it operates on are also large. Second, the ease with which we could perform the comparisons, and look 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



Figure 4: The lowest layer of the temperature tendency field, as computed by HIRPBL, before (left) and after (right) modifying HIRPBL to apply 13 iterations at all columns.

knowledge of the structure or operation of MM5. In all, it required about 1 hour of time from the initial examination of the code through to the specification of the assertions until the results were delivered.

5.2 Comparing Parallel and Sequential Codes

In the second experiment reported in this paper, we used Guard to compare the sequential MM5 and a 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.

A preliminary set of assertions applied to major data structures showed a number of differences. The differences were initially small and grew slowly. Visualization showed significant spatial structure, suggesting that they 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 4).

Having localized the error to the HIRPBL routine, we switched to a manual code inspection. We soon identified the source of the discrepancy. Briefly, the problem turned out to be a difference between the way 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.

For maximum vectorization, 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 communication (to determine the maximum number of iterations) and work. Hence, the parallel code performed just the required number of iterations for each column.

Suspecting that this difference in behavior was the source of our problems, we modified both the sequential and parallel codes so that both performed 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 4). 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. Some divergence remains, almost exclusively around the second row and column in from the boundary. There is also a small secondary discrepancy near the northern boundary. As time progresses the influence of the divergences spreads out in a predictable fashion, but the magnitude remains small. We believe that these divergences reflect two additional errors that are, as yet, unaccounted for.

In summary, in this experiment Guard allowed us first to account for one significant discrepancy between the sequential and parallel codes, and second to identify two other differences, to be located in further investigation.

5.3 Guard Extensions

The second experiment also served to motivate some future extensions to the Guard debugger. As noted above, MM5 and MPMM have very different code and data structures. Guard was able to deal with the different data layouts quite naturally, using array index permutations and subarray expressions. However, one aspect of the code caused it difficulty. MM5 tends to apply code blocks to entire rows of the data grid before proceeding to the next code block. In contrast, MPMM tends to invoke much larger code blocks on one grid point at a time. A consequence of these differences in structure is that it becomes difficult to compare array values at an arbitrary point in time. The following simplified example shows why this is so.

MM5: do i=1,nlon x(i) = x(i) + 2 enddo do i=1,nlon y(i) = x(i) + y(i) enddo do i=1,nlon x(i) = x(i)/y(i) enddo	MPMM: do i=1,nlon x(i) = x(i) + 3 y(i) = x(i) + y(i) x(i) = x(i)/y(i) enddo	<< 1. COMPARE x << 2. COMPARE x << 3. COMPARE x
--	---	---

The code fragment on the left has an MM5-like structure; for simplicity, code blocks are single assignments. The code on the right has an MPMM-like structure; the code block in the case consists of three assignments. Observing that array **x** is incorrect after execution of this code, we want to determine which of the two assignments to **x** introduces an error. (In this case, the first assignment turns out to be the culprit.) Hence, we want to compare values of **x** before this code fragment and after each assignment. The three comparison points are marked. The first and the third comparisons can be performed easily, but the second cannot. In the MPMM-like structure, the first assignment to a single element is always followed by a second assignment to the same element. The entire array **x** is never in the state of having only the first assignment applied to all elements, but not the second.

Currently, the MPMM program code has been modified to store intermediate values in a temporary array, which is used for comparison at the end of the first **do** loop in MM5. Extensions to Guard, which will automatically handle such temporary arrays, are planned for the future.

6 Conclusions

Relative debugging is a new debugging paradigm for applications which undergo evolutionary changes; Guard is a debugger that supports this paradigm. 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 have 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 which will significantly increase its utility is support for the debugging of parallel programs. Another direction, motivated by the MM5 experiments described in this paper, is to provide support for more flexible comparison of codes with different control structures.

Acknowledgments

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.

The paper reports a collaborative project between Griffith University and Argonne National Laboratory. Guard was developed at Griffith University during 1994. The experiments reported in the paper 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.

References

- [1] R. Anthes. 1986 summary of workshop on the NCAR Community Climate/Forecast models. *Bull. Amer. Meteor. Soc.*, 67:194–198, 1986.
- [2] Z. Aral, I. Gertner, and G. Schaffer. Efficient debugging primitives for multiprocessors. In *Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 87–95. ACM, 1989.
- [3] J. Brown and K. Campana. An economical time-differencing system for numerical weather prediction. *Mon. Wea. Rev.*, (106):1125–1136, 1978.
- [4] J. Dudhia. A nonhydrostatic version of the Penn State/NCAR mesoscale model: Validation tests and simulation of an Atlantic cyclone and cold front. Technical report, National Center for Atmospheric Research, 1992.
- [5] I. Foster and J. Michalakes. MPMM: A Massively Parallel Mesoscale Model. In Geered-R Hoffmann and Tuomo Kauranne, editors, *Parallel Supercomputing in Atmospheric Science*, pages 354–363. World Scientific, River Edge, NJ 07661, 1993.

- [6] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings Supercomputing-93, Portland, Oregon*, pages 462–471. IEEE, 1993.
- [7] Georg A. Grell, Jimmy Dudhia, and David R. Stauffer. 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.
- [8] J. H. Griffin, H. J. Wasserman, and L. P. McGavran. A debugger for parallel processes. *Software-Practice and Experience*, 18(12):1179–1190, December 1988.
- [9] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.
- [10] J. Michalakes, T. Canfield, R. Nanjundiah, S. Hammond, and G. Grell. Parallel Implementation, Validation, and Performance of MM5. In *Parallel Supercomputing in Atmospheric Science*. World Scientific, River Edge, NJ 07661, 1994.
- [11] T. G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.
- [12] R. A. Olsson, R. H. Crawford, and W. W. Ho. A dataflow approach to event-based debugging. *Software-Practice and Experience*, 21(2):209–229, February 1991.
- [13] E. Satterthwaite. Debugging tools for high level languages. *Software-Practice and Experience*, 2(3):197–217, July-September 1972.
- [14] T. Shimomura and S. Isoda. Linked-list visualization for debugging. *IEEE Software*, 8(3):44–51, May 1991.
- [15] R. Sosič. Design and implementation of Dynascope, a directing platform for compiled programs. *Computing Systems*, to appear, 1995.
- [16] R. Sosič. A procedural interface for program directing. *Software-Practice and Experience*, to appear, 1995.
- [17] P. T. Zellweger. Interactive source-level debugging of optimized programs. Technical Report CSL-84-5, Xerox PARC, 1984.