

Assertion Based Parallel Debugging

Minh Ngoc Dinh, David Abramson, Donny
Kurniawan, Chao Jin

Faculty of Information Technology
Monash University

Clayton, 3800, Victoria, Australia
{ngoc.minh.dinh, david.abramson, donny.kurniawan,
chao.jin}@monash.edu

Bob Moench, Luiz DeRose

Cray Inc

Saint Paul, MN 55101-2987, United States
{rwm, ldr}@cray.com

Abstract—Programming languages have advanced tremendously over the years, but program debuggers have hardly changed. Sequential debuggers do little more than allow a user to control the flow of a program and examine its state. Parallel ones support the same operations on multiple processes, which are adequate with a small number of processors, but become unwieldy and ineffective on very large machines. Typical scientific codes have enormous multi-dimensional data structures and it is impractical to expect a user to view the data using traditional display techniques. In this paper we discuss the use of debug-time assertions, and show that these can be used to debug parallel programs. The techniques reduce the debugging complexity because they reason about the state of large arrays without requiring the user to know the expected value of every element. Assertions can be expensive to evaluate, but their performance can be improved by running them in parallel. We demonstrate the system with a case study finding errors in a parallel version of the Shallow Water Equations, and evaluate the performance of the tool on a 4,096 cores Cray XE6.

Keywords: *parallel debugger, assertion, Guard, MPI.*

I. INTRODUCTION

While programming languages have advanced tremendously over the years, program debuggers have hardly changed. Existing debuggers do little more than allow a user to control the flow of a program, and examine its state. More dramatically, parallel debuggers allow the user to control multiple independent threads of execution but otherwise expose the same functionality found in their sequential counterparts. Perhaps this was unimportant when both the program state and the size of the parallel machines, was small. However, modern scientific codes manipulate enormous data structures and run on thousands of cores.

We see two main sources of challenge in current approaches to debugging parallel applications. First, the state of a typical scientific application can be very large, making it infeasible for users to examine individual cells of, what are typically, large multi-dimensional floating point structures. It simply isn't feasible to print out complete arrays to decide if a program is computing the correct results. We refer to this as a *cognitive* challenge, because programmers cannot integrate the data effectively into a mental model of correct execution. The most powerful technique available for integrating large amounts of data into a human visible form

is to use scientific visualisation tools [1]. This approach has been used to find programming errors in scientific applications, but in general, it is not always possible to “see” the errors in the data. Second, whilst parallel debuggers are used to *debug* parallel programs, they generally don't make use of the underlying parallel platform to improve their *own* performance. Thus, debuggers are typically restricted to techniques that can be executed sequentially on the front-end node in a reasonable time. Recently, there are notable works, which focus on formal and semi-formal verification of MPI concurrency and message flow checking [23][24]. However, we only focus on the challenges addressed above. In this paper, we envisage a set of powerful debugging primitives that diminish the cognitive challenge, but these would normally be too expensive to run sequentially. However, by using the inherent parallel processing capabilities of the supercomputer (which is usually idle when a program is suspended at a breakpoint), a range of more powerful debugging operations are feasible.

Fortunately, research in program proving and verification provides a fertile base on which to build powerful debugging primitives. Many years ago, Dijkstra [2], Hoare [3] and others introduced the idea of program assertions - predicates that are evaluated as a program runs to indicate whether an error has occurred. Assertions in the form of pre and post-conditions are now routinely used in production codes, and are exemplified in languages such as Eiffel [4]. Assertions are usually implemented as additional code that is inserted into the base program at compile or link time, making it possible to check these, for example, on the entry and exit of a procedure. However, language based assertions are not meant for debugging per-se but rather as sanity checks on the state of the program.

In this paper we argue that debug-time 'ad-hoc' assertions are a viable way of addressing the cognitive challenge in debugging large scientific codes. Such assertions can test the state of a large distributed array, for example, and can be refined iteratively by the user in order to locate the source of an error. However, because the evaluation of such assertions is likely to be expensive, we propose a scheme that executes the assertion in parallel, making assertions over large data structures feasible. We discuss the implementation of the ideas in our parallel debugger called Guard [7].

II. ASSERTIONS

An assertion is a statement about an intended behaviour of a system's component, which must be verified during execution. In software development, a programmer defines an assertion to ensure specific program state at run time. Hence, programmers can inject design intentions, such as constraints or contracts, into the code. Assertions have been used extensively for evaluating invariants, checking input parameters, and for enhancing program correctness and quality.

A. Traditional Programming Language Approaches

The key features of assertions are *error detection*, *error isolation*, and *error notification* [8]. Using assertions whilst developing software deliver various benefits. First, embedding assertions in the code improves the *observability* within the design [8]. With assertion mode enabled, as soon as an assertion violation occurs, the programmer can immediately use the information provided by the assertion failure to identify the defect. Second, using assertions enhances the integration between different modules through correct usage checking. Finally, as the result of observability improvement, using assertions effectively reduces the debugging time.

B. Debug-Time Assertions

Whilst the programming language community has focussed on adding assertions to programs, they can also be used as a powerful tool during the debugging phase. When used in this way, a programmer writes 'ad-hoc' assertions at debug time to test the state of a running program and try locating errors. However, while conceptually appealing, very few debugging environments support assertions.

Siegel et al. [22] introduce a novel type of assertion called *Collective Assertion* to address the importance of evaluating assertions in parallel programs. Collective assertion asserts the global state of the parallel program by taking snapshots of processes' state and delivers them to the *Toolkit for Accurate Scientific Software* (TASS). TASS uses symbolic execution and explicit state enumeration techniques to evaluate the assertion given a set of local state snapshots. Nevertheless, because TASS executes sequentially, bottleneck can be formed when the size of the problem as well as the machine's size scale up. The authors only present performance evaluation for up to 16 processes [22].

In related work, a plug-in is developed to assist CharmDebug [6], a specific debugger for the Charm++ parallel programming language, regarding runtime data introspection. The plug-in allows the user to upload specific Python scripts to the runtime parallel applications (referred as servers) and execute them in order to perform data checking for debugging purposes. The aim of this work is close to what we are trying to achieve. However, since there is no data decomposition information attached with the Python script, it is not clear how the actions can be applied on the overall data structure, which is distributed across

many processing units. Likewise, TotalView [5] supports debugging scripts, in a language called *TVScript*, which are executed at breakpoints. *TVScript*, supports actions such as displaying variables, displaying a back trace, showing the memory allocation, evaluating potential memory leaks, and generating logs for post-mortem analysis. However it is difficult to reason about the collective state of an entire program because scripts only access individual processes. Thus, it is difficult to implement an assertion such as "the sum of all elements of array A is to be less than X" without building multi-process aggregation code.

In a similar vein, GDB [9], allows developers to specify *conditional breakpoints*. A conditional breakpoint is a simple breakpoint, triggered when the attached condition fails at the specific location in the source code. Using conditional breakpoints, users can assert the state of variables at runtime. In addition, GDB enables user-defined function which can effectively be used to calculate, test, and print data variables accessible from GDB. The combination of conditional breakpoint and user-defined functions is a promising tool for monitoring the state of the application efficiently. Nevertheless, GDB is a sequential debugger, and does not support debugging parallel programs. Thus, it is not possible to write conditional breakpoints that reasons across the state of multiple processes. Moreover, since it has no way of describing a globally distributed data structure, it is not possible to write an assertion about such a structure.

Other debuggers, such as p2d2 [10] and DDT [11], specifically written for parallel machines also do not support assertions on global data structures because they do not have an understanding of the way the compiler, or the user, has decomposed the structure. As a result, there is no easy way to write an assertion that reasons about the state of a global array that happens to be decomposed across multiple processors.

One debugger that does provide some of these properties is Guard, a parallel tool that supports a concept called relative debugging [7]. In relative debugging, the user writes assertions that compare the contents of data structures in one program with those in a similar reference program. These are used to find out where a new program diverges from an existing one, and has applicability in programs that evolve and are modified over time. However, Guard's assertions only test for equality of structures between two different programs, and thus are of limited use when there is only one running program.

In this paper, we discuss how we have leveraged the existing Guard run-time machine, and have implemented ad-hoc assertions that work on a *single parallel program*. Like other parallel debuggers, Guard supports the control of a multi-process parallel program, using extensions to conventional sequential debugger primitives. Specifically, it uses a command line syntax called HPDF that supports concepts such as parallel program invocation, process sets and the like [7]. Over and above this, Guard supports:

- a data decomposed function that defines how a given array has been distributed across parallel memories;
- parallel expression evaluation in debug servers; and
- assertions that reason about distributed data structures.

These three features make it possible to build and execute ad-hoc assertions and use them to test and debug programs that fail. Importantly, the work reported here extends our previous work by allowing Guard to execute assertions on single parallel programs in a scalable way.

C. A Template for Ad-hoc Assertions in Guard

Guard is primarily a command line debugger (although we have built a number of GUI based versions, including an integration with Eclipse [12]). Its command syntax is based on a standard proposal called HPDF, which supports standard ‘GDB like’ commands such as *breakpoints*, *print*, *step*, and so forth.

1) Guard’s Commands

When a user starts a parallel program using the *invoke* command, Guard returns an array of process identifiers as a process set (*<process_set>*). Process sets can be used to alter the focus of any commands that are issued, thus a user can restrict, or broaden, any command to a range of processes. Process sets can also be used by an *assert* statement. An assert statement contains a source file (*<source_file>*) and a line number (*<line_num>*) to indicate where the assertions are to be evaluated.

2) Assertion Syntax

A commonly used assertion consists of a predicate built by two or more *expressions*. An expression in Guard is defined using the following grammar:

```
E ::= E <compute_op> E
      | constant
      | proc_set::<gdb_var_expr>@<source_file>:<line_num>
<compute_op> ::= [+,-,*,/]
<compare_op> ::= [=,>,<,>=,<=]
```

An assertion is therefore defined as:

```
assert E1 <compare_op> E2
where:
```

E₁ and E₂ are constructed by the grammar above

Note that *<gdb_var_expr>* can be any acceptable GDB expression, which means it could be complex when a pointer to an array is to be used or type-casted.

3) Supporting Reduction Operations

Guard supports a set of reduction functions including *sum*, *product*, *min*, *max*, and *hash*, which can be applied, in parallel, on a set of data values. Users can specify which type of reduction function can be exercised for different assertions according to their nature. To enable a reduction function, Guard provides the ‘*set reduce*’ command. Below is a simple example of how to trigger the *sum* reduction function in Guard.

set reduce “sum”

4) Data Decomposition

Importantly, for the debugger to evaluate an assertion on a distributed data structure, it must be aware of the way that data is decomposed across the processors. For example, we may wish to assert that every element of a 2D distributed array, *var*, is *> 50,000*. To evaluate this assertion, Guard requests the individual debug servers to extract the portions of *var* that they own, perform comparison of each element against 50,000 and return an array of values to the debugger client. The debug client then needs to know the decomposition scheme used in order to rebuild the complete result array and display to the user as the outcome of the assertion.

In order to describe the data decomposition, we implement a special function in the debugger called *blockmap*. Blockmap is inspired by the *block-cyclic data distribution* scheme [13], and follows the HPF syntax [14]. It includes a collection of expressions that describe the decomposition scheme. Consider the example below:

```
blockmap test(P::V)
  define distribute(block, *)
  define data(80,160)
end
assert test($a::var@“par.c”:100) > 50,000
```

We first define a blockmap function named *test* which describes that a 2D array of size (80,160) is distributed as blocks of rows, followed by an assertion using the *assert* command. When the assertion is executed, for each of the running process in the process set *\$a*, data from array *var* is collected at line 100 in source file *par.c*. Each element in the array is compared against 50,000 to generate a collection of boolean results. On the client side, these result collections are aggregated into one big overall result array using the blockmap *test* function.

D. Examples of Ad-hoc Assertions

Here we specifically consider potential assertions that are useful for debugging parallel supercomputing applications. Whilst there is no single template for a parallel program, we make three assumptions that underpin our current implementation.

First, scalable applications achieve their performance by dividing the problem, and the program state, across multiple processes. Each process manipulates and transforms the data as independently as possible in order to minimize synchronization overheads. Typical examples are partial differential equation (PDE) solvers in which the key multi-dimensional data structures are distributed across the processors, and the state equations transform the contents of the variables as the algorithm proceeds. Because these solvers march forward in discrete time steps, it is possible to make assertions about the state variables at each iteration.

Second, typical message passing programs are composed of a single master process and a set of slave processes (often statically created). We assume that the master communicates with the slaves infrequently (the master may

do little more than launch the slaves and wait for them to complete, and in fact, it may actually be one of the slaves), but slaves are free to communicate with each other, to synchronize and share state within iterations.

Third, array data types are first class and can be used in assertion expressions. Moreover, we support operators, such as array *slicing*, that work over sub-arrays and orthogonal sub blocks, making it possible to select sections of complete arrays.

Here we consider some use case scenarios for assertions, and later in the paper, we illustrate their use in debugging a scientific parallel application. Note that it is unlikely that we would use all templates in any one debugging session, and they are used here to demonstrate the range of possible debugging operations. Whilst it is possible to implement some of these assertions in existing parallel debuggers, in general, they are not supported.

1) *Assertions about Master State*

Here we wish to make statements about the state of the master process, typically to test the initial and final conditions. Moreover, if slaves communicate with the master during the execution, these assertions can test the global state of the system as perceived by the master. These assertions can be arbitrary array expressions that evaluate to true or false. For example, we might assert that all elements in a two dimensional array are > 0 at a particular line of the code. Alternatively, array slicing operators can be used to select sections of complete arrays.

2) *Assertions about Slave State*

These assertions are similar to the master state ones, but apply to either the entire set, or a restricted subset, of slaves. For example, we may assert that a particular variable in each slave must be < 0 , or that all elements of an array in each slave must be > 0 . The assertion fails if *any* of the slaves' assertions fail. Notably, these assertions treat the state of each slave as independent of any other, and thus are not used to reason about global data structures that have been distributed across the processors.

3) *Comparing State between Slaves*

Here we wish to compare variables in one slave with variables in another. For example, we might wish to check that slaves are correctly sharing data in ghost bands of arrays. Such assertions would allow us to check that the inter-slave communication has been performed correctly.

4) *Comparing Master and Slave State*

Here we wish to compare variables in the master with slave variables. For example, we might distribute a set of constants to all of the slaves, and then want to check that they have been initialized correctly. In a more complex example, a computation might distribute a data structure across the slaves that each manipulates only a portion of the original array. In this type of assertion, we wish to check that the data has been correctly distributed to the slaves. In order to do this, the debugger needs to know the data decomposition scheme used so it knows which elements to

compare, and needs to support the concept of data distribution.

5) *Reductions over Slaves State*

These assertions are similar to 2.4.4, but allow us to apply a reduction operator across the slaves. For example, we might want to take the average, or standard deviation, of values in the slaves and test the reduced result against a threshold. Or, we might want to compute some reduction operator across a distributed array, involving both the data decomposition functions as well as global reductions.

It is clear that in some of these assertions, we can leverage the underlying parallel machine to improve the performance. In particular, reductions can be computed in parallel during the computation phase, and then reduced using tree based reduction operators where possible.

III. IMPLEMENTATION IN GUARD

Here we briefly describe the architecture of our existing relative debugger, Guard. Numerous articles discuss the use of relative debugging, which is not the focus of this paper [7, 15]. However, we have leveraged the infrastructure to build a proof of concept assertion debugger, and thus we describe Guard in sufficient detail to illustrate the implementation.

A. *Guard Architecture*

Guard employs a client/server model, where the debugger is divided into a single client and multiple servers, to ensure that the processes being debugged can be distributed onto multiple platforms and can be controlled independently (Figure 1). Other parallel debuggers, such as TotalView, P2D2 [10], and DDT, have adopted a similar client/server architecture, albeit with different server technology. For example, DDT uses GDB as its debug engine, whereas TotalView has its own proprietary code.

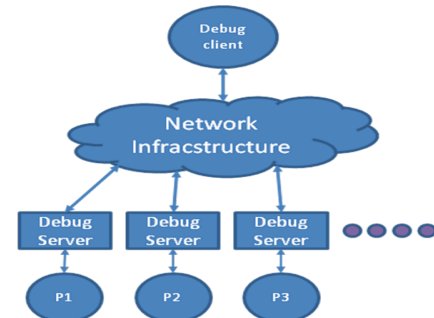


Figure 1 - Client/Server Architecture

The debug client is the point at which the user interacts with the debugger, so it is generally executed on the user's local host. The client user interface accepts and processes user commands in either of two modes: immediate or deferred. Deferred commands are compiled into a *dataflow graph*, which is then executed by a runtime *interpreter*. This is the core engine for running assertions in relative debugging, and has been used to implement single program ad-hoc assertions as discussed in this paper. The following sections discuss the implementation of the dataflow compiler and dataflow engine that support the evaluation of

multiple ad-hoc debug-time assertions. There are other important architectural aspects of Guard which interested readers can find full description in other literature [7, 15]. The Network Infrastructure supports different communication protocols including basic socket connections and tree based communication networks such as MRNet (Multicast/Reduction Network) [16].

1) Dataflow Compiler

Executing the proposed assertions requires the debugger to handle asynchronous events. Consider the example of an assertion that requests the comparison of two variables at different breakpoints. With control-flow semantics, the debugger can be deadlocked while waiting for both breakpoints to be reached. Therefore, we employ a dataflow execution mechanism to manage the process of capturing runtime data. A full description of this idea is found in [7]. A dataflow compiler translates assertions into a low-level graph description. The graph is later used by the *dataflow engine* to execute the assertions at runtime using an execution semantics much like early dataflow computers [17].

Below is an example of an abstract dataflow graph generated for the assertion beneath.

set reduce "sum"

assert \$a::p_array@"par.c":34 > 1,000,000

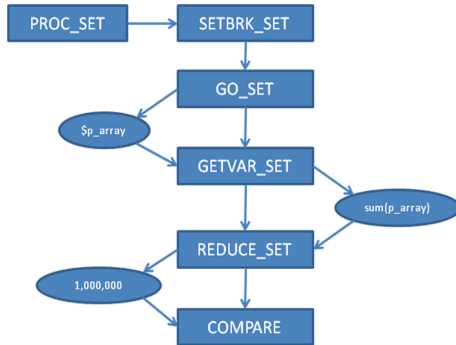


Figure 2 - Abstract Assertion Dataflow Graph

2) Graph Interpreter

When user starts the graph, graph interpreter operates the nodes sequentially. In Figure 2, oval nodes denote data tokens while rectangular nodes represent activities and commands. Activity nodes receive data tokens as inputs and output data tokens to other activities accordingly. We can categorize activity nodes into two types: local command nodes such as COMPARE and debug server command nodes such as DBGGO_SET and GETVAR_SET. With debug server commands, the graph interpreter packs the command into a request and broadcasts that to the debug servers. Debug servers then translate the request into low-level debug commands (such as break, print or continue) and pass them to GDB instances (We have used GDB as a matter of convenience and could replace this with any equivalent debug engine).

In the example above, the assertion requests data collected from variable *p_array* to be reduced using the sum

function, and the result is compared against the value 1,000,000. Figure 2 describes the process of setting breakpoint given from the assert statement on the process set. The program is executed until the breakpoint occurs. The GETVAR_SET node instructs the debug servers to extract data from *p_array* using the GDB's *print* command. The data is then reduced using *sum* function and further reduced at the client REDUCE_SET node. This approach allows the debug servers to work independently on their own copy of the sub-datasets in parallel. Finally, the outcome of the assertion is performed by the COMPARE node.

B. Debug Server

When implementing ad-hoc assertions of the type discussed in this paper, we have extended the debug servers used in earlier version of Guard [7]. In particular, remote debug servers now execute the following operations: data retrieval, data reduction, and data collection to client node.

1) Data Retrieval

In Figure 2, 'GETVAR_SET' instructs the debug servers to return data from variable *a*. Note, the 'GETVAR_SET' node is only executed when all of the application processes are already at their barriers (i.e a breakpoint).

2) Data Reduction

The debug server performs data reduction functions before the result is transferred back to the client. We deliver extra information to the debug servers, including the reduction function's ID and data decomposition information. Upon accepting the request, remote debug servers have sufficient information to perform the reduction operation in parallel and transmit the reduced dataset back to client. We currently support a range of reduction functions including sum, min/max, average, and hash [17].

3) Data Collection

After completing the reduction operation, debug servers transfer the results to the client. Different assertions require different handling in the client and servers. For example, in Figure 2, since the assertion calculates the sum of all elements in array *var*, upon receiving sub-total values from remote debug servers, the client performs the final reduction sum before comparing the result to 1,000,000. In this case, we can assert the sum of all elements in array *var* to be greater than 1,000,000 even though *var* has been decomposed across many processing units.

IV. CASE STUDY: SHALLOW WATER EQUATIONS PROGRAM

In this section we illustrate the use of different assertion types, discussed in section 2.4, for finding errors in a simple, but scalable, scientific application, the Shallow Water Equations [18]. It is a vastly simplified version of the code found in real weather models, such as WRF [19], but is simple enough to explain in a small case study. We believe this demonstrates the effectiveness of the technique without getting lost in the detail of real codes.

A. The Shallow Water Equations

The shallow water equations describe the motion of an incompressible fluid with a free surface, with the constraint that the horizontal scales of motion are much larger than the vertical. For more details see [18].

The shallow water equations are a favored choice for experiments with various model structures and numerical schemes. Although a very simple representation of the atmosphere, they do include the two types of horizontal wave motion important in more realistic models, gravity waves and Rossby waves. They are also useful for experiments in parallelizing because they include much of the structure (and so problems with data communication and synchronization) of more complicated models.

Many years ago, we implemented a MPI program for the finite difference form of the equations [20], and have reused this in this case study. In this code a master process initially distributes the key variables to each of the slaves, and these move through a sequence of steps. At each step, the slaves interchange part of their state variables with neighbors. At the end of a time step, they independently step forward, and the whole process continues.

B. Debugging Shallow Water Program

1) Assertion 1 – checking master state

Shallow Water program begins with the initialization of key data structures before any time step computation commences. The initialization process happens in two phases: initializing key data structures in the master process and distributing those to the compute slaves.

The following assertion checks that the pressure array in the master is correctly initialized before any calculations are undertaken.

```
assert $m::p@"main.c":180 = 50000.000
```

Guard sets a breakpoint at line 180 in *main.c*. When the breakpoint is hit, it extracts the content of *p* (the pressure array) from *\$m* (the master process) and checks that each array element equals the constant 50,000. The violation of the assertion reveals an error in the initialization code, as shown below. As a result, the assertion indicated that there were multiple elements in *p* not equal to 50,000. Inspection revealed an indexing defect, namely only the diagonal is initialized:

<pre>for (j = 0; j < n; j++) { for (i = 0; i < m; i++) { p[j][j] = 50000.;}} Incorrect code</pre>	<pre>for (j = 0; j < n; j++) { for (i = 0; i < m; i++) { p[j][i] = 50000.;}} Correct code</pre>
---	---

Rerunning the assertion after fixing the index value shows pressure array values in master process were successfully initialized.

2) Assertion 2 – checking communication between the master and slaves

During the second phase of the initialization process, the pressure array *p* is distributed to the slaves using a (*block, **) decomposition. In this case we were trying to find an error

in the communication code responsible for the slave array initialization.

The following assertion checks that each slave receives the right portion for its work.

```
blockmap test(P::V)
  define distribute(block, *)
  define data(1024, 1024)
end
assert test($a::p@"main.c":21)=$m::p@"main.c":43
```

Blockmap is a decomposition definition. With it, the debugger can rebuild the global data structure from the distributed pieces. In this example, the function *test* describes the decomposition of a two dimensional array of size 1024x1024 into blocks of rows. The parameters *P* and *V* are placeholders for the set of processes (*P*) and the variable (*V*).

The assertion checks that the decomposed array, *p*, in the slaves (*\$a*) is the same as the *p* array in the master process (*\$m*). It uses the function *test* to know how to recombine the pieces into a single array.

Execution of the assertion indicated that some of the slaves were not initialized correctly. Closer investigation of the MPI Send code from the master process revealed an error. In particular, where data in *p* array is sent to the slaves, the bounds of the inner *j* loop are incorrect. Correcting this code fixed this bug.

<pre>for (i = 1; i < proc_cnt; i++) { for (j = 0; j < cs; j++) { acopy_two_to_one(p, p_start, j) MPI_Send(&p_start, m, MPI_FLOAT, i, P_ROW, MPI_COMM_WORLD);}}</pre>	Incorrect code
<pre>for (i = 1; i < proc_cnt; i++) { for (j = (i-1)*cs; j < cs+((i-1)*cs); j++) { acopy_two_to_one(p, p_start, j) MPI_Send(&p_start, m, MPI_FLOAT, i, P_ROW, MPI_COMM_WORLD);}}</pre>	Correct code

3) Assertion 3 – checking slave to slave communication

Before and after each major computational step shown, each slave synchronises the edges of the data structures with its immediate neighbours, using the following code:

```
/* loop over latitudes calculating U, V, z and h
   do the block of latitudes from jstart to jend */
load_ghost(
  prv, nxt, my_id, jstart, jend, p, u, v, p_sync,
  u_sync, v_sync);
calcuvzh(
  jstart, jend, p, u, v, cu, cv, h, z, p_sync,
  u_sync, v_sync, cu_sync, cv_sync, h_sync,
  z_sync, fsdx, fsdy);
unload_ghost(
  prv, nxt, my_id, jstart, jend, cv, z,
  cv_sync, z_sync);
```

Each slave needs to communicate with two neighbors to perform the synchronization process. In order to successfully synchronize data between slaves, given that each slave only carries a portion of the overall data structure, a ghost band is required. Accordingly, *load_ghost* receives a ghost band from one neighbor, and sends information to the ghost band of the other neighbor. Likewise, *unload_ghost* sends information in the other direction.

The following assertions test the ghost band synchronization for the pressure variable, *p*, and its corresponding ghost band, *p_sync*, between the three of the slaves:

```
assert $a[0]::p_sync@"main.c":448 =
      $a[1]::p[0][0..1024]@"main.c":448
assert $a[1]::p_sync@"main.c":448 =
      $a[2]::p[0][0..1024]@"main.c":448
assert $a[2]::p_sync@"main.c":448 =
      $a[3]::p[0][0..1024]@"main.c":448
```

The outcome of the assertion shows that *p_sync* variable from process *\$a[0]* is not the same with 1st row of *p* array from process *\$a[1]*. *load_ghost* contains the following code:

```
load_ghost(
  prv,next,my_id,jstart,jend,p,u,v, p_sync, u_sync,
  v_sync)
{
  acopy_two_to_one(p, tmp_ds, 0);
  neighbour_send(prv, my_id, CALC1a, tmp_ds);
  acopy_two_to_one(u, tmp_ds, 0);
  neighbour_send(prv, my_id, CALC1b, tmp_ds);
  acopy_two_to_one(v, tmp_ds, 0);
  neighbour_send(prv, my_id, CALC1c, tmp_ds);
  neighbour_receive(next, my_id, CALC1b, p_sync);
  neighbour_receive(next, my_id, CALC1a, u_sync);
  neighbour_receive(next, my_id, CALC1c, v_sync);
}
```

For each *neighbour_send* and *neighbour_receive* function call, there is an ID value passed to indicate which key data structure is to be synced. For example, *CALC1a* means pressure array *p*. Executing the assertions shows that *p* array was not synced successfully. Examining the body of *load_ghost* function shows that wrong ID values was passed between *u* and *p* arrays on the receive ends. In this example, each of the assertions is executed in parallel because the data flow graph contains independent threads of execution.

<i>neighbour_receive</i> (next, my_id, CALC1b, p_sync); <i>neighbour_receive</i> (next, my_id, CALC1a, u_sync);	Incorrect code
<i>neighbour_receive</i> (next, my_id, CALC1a, p_sync); <i>neighbour_receive</i> (next, my_id, CALC1b, u_sync);	Correct code

4) Assertion 4 – Checking slave state across time steps

The formulation of the shallow water equations used in this paper is based on a finite different approximation.

Because of this it is possible to calculate a bound on the changes in any of the array values within a time step, which are limited by the time step increment, the finite difference step and the range of values¹. This information can be used to set a bound on the difference between old values of each of the state arrays and new values, and this can be implemented as part of an assertion that checks the solution as new values are calculated.

During execution, the application maintains both current pressure values in *pnew* array and previous pressure values in *pold* array. The following assertion checks that the difference is less than an upper bound.

```
assert $a::pnew@"main.c":463
      $a::pold@"main.c":463 < A
```

Because of an error in the time step code, we detected a difference that was out of bounds. The result was, again, an indexing error, as shown below.

<i>pnew[i][j]=pold[i][j]+tdt*dpdt[j][j];</i>	Incorrect code
<i>pnew[i][j]=pold[i][j]+tdt*dpdt[i][j];</i>	Correct code

5) Assertion 5 – Reducing slave state

Finally, a deeper understanding of the physics of an application can be used to specify very powerful assertions on the state of the system. For example, the shallow water equations conserve both mass and energy. This means that as the calculation proceeds, the total amount of mass in the system should only change a small amount (subject to numeric resolution), and any divergence on this indicates that the program is not behaving correctly. Using this type of information, we could write an assertion which tests whether mass and energy are indeed conserved. While our formulation of shallow does not directly compute the mass, there is another invariant in the total pressure in the system. Thus, an assertion that checks the total pressure in the system is useful for detecting errors. The following commands perform a reducing sum across the pressure from all slaves, and checks whether this is within a predefined error bound.

```
set reduce "sum"
set error 1e-6
assert $a::p@"main2.c":463 = 50,000
```

Upon execution, Guard actually computes a set of partial sums in parallel, and then reduces these using a further sum on the client side. “*set error*” statement sets an error tolerance to 10⁻⁶.

In this example, Guard executed the code for a number of cycles, and confirmed that the total pressure was within bounds. As a result, we concluded that the code was working correctly.

¹ We actually observed the upper bound used here in a version of the code that was working correctly, rather than calculating it from first principles. However, the latter is possible if a correct version of the code is not available.

6) Summary

In this case study, we presented a debugging course using our tool to detect, locate and fix several bugs in the Shallow Water Equations program. Even though the debugged application is relative small and simple comparing to real scientific applications, it serves well the purpose of illustrating the kind of assertions that we are promoting. This case study should be used as a detailed example of how to debug parallel applications using ad-hoc debug time assertions. In the following session, we benchmark the performance of our proposed tool.

V. PERFORMANCE ANALYSIS AND EVALUATION

In this section, we evaluate the performance of the sum assertion used in section 4.2.5 on a Cray XE6 Gemini 1.2 system, with 752 compute nodes, totaling 17,472 cores. We secured 4,096 cores for our experiments.

To demonstrate that the approach is applicable to large-scale scientific codes, we use information about data structure sizes in WRF, a production climate model [19]. In this configuration, key variables contain 4,486 by 4,486 horizontal grid points and 101 vertical levels (2 billion single precision floats). When run on a parallel machine, this is decomposed across the processors using similar techniques discussed in section 2.3.4. Using these parameters, we created both strong and weak scaling experiments. These are discussed in detail in sections 5.1 and 5.2.

A. Strong Scaling Experiment

In strong scaling, the size of the overall data structure remains constant as the number of processors changes (i.e 4,486x4,486x101 floating point values); thus data distributed to each worker reduces as the number of processes increases.

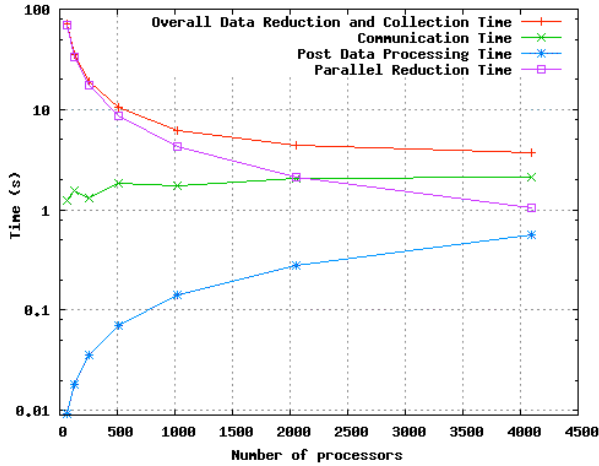


Figure 3 - Strong Scaling Assertion Evaluation Time

Evaluating an assertion consists of a few phases, as described in section 3.2. Of these, data reduction and data collection consume most of the time. First, after receiving the command from the debug client, the remote debug servers obtain data from GDB, and perform the requested data reduction operation. This step proceeds in parallel since

each debug server works independently. Second, through the network infrastructure, data is reduced, collected and returned to the debug client. We denote the time spent on this step as the *communication time*. Finally, the client executes *post data processing* tasks, performing the final reduction sequentially.

Figure 3 shows these times on a log scale. As the number of processors increases, the *parallel reduction time* falls as expected. The *post data processing time* grows as the number of processors increase, because it is performed sequentially in the client, however, it is very small compared to the other times, and does not affect the speedup. The *communication time* rises very slightly. More importantly, it is around one second, which is significant compared to the *reduction and communication times* at 4,000 cores. This is most likely the communication start up latency for the MRNet middleware, which is tuned for large-scale network communication, and uses socket connections between nodes. This limits the speedup at around 1,000 cores, as indicated in Figure 4. Nevertheless, the tool achieves reasonable speedup overall, and more importantly, reduces the assertion execution time to the order of seconds, making it feasible as an interactive debugging aid. We expect to be able to improve this time through tuning the MRNet code.

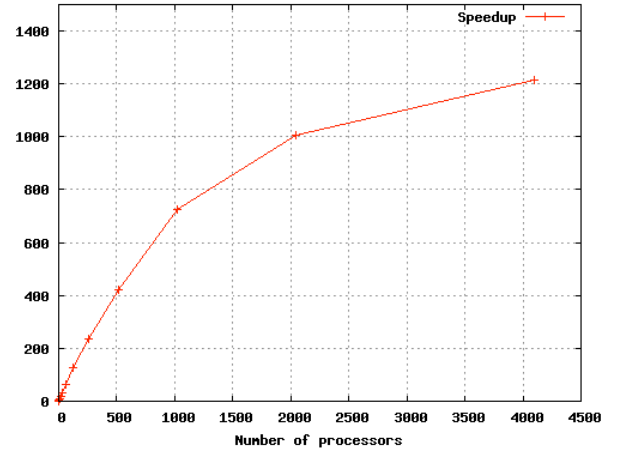


Figure 4 - Assertion speedup against #processors

B. Weak Scaling Experiment

Weak scaling keeps the data assigned to each compute node constant; thus the total amount of data grows as the number of processors increases.

In this experiment, as we increase the number of processors (i.e more data involved), the assertion evaluation time rises very slowly, as shown in Figure 5. The parallel reduction time remains fairly constant in this experiment. Therefore, it is the sum of extra communication time and post data processing time that grows. However, since this time does not grow linearly, the curve starts to flat out at 512 cores. This implies that given more compute resources, debugging larger applications with our tool is still practical.

Based on these results, we believe the execution of our adhoc debug-time assertions returns acceptable response

time for both weak and strong scaling of parallel codes on large machines.

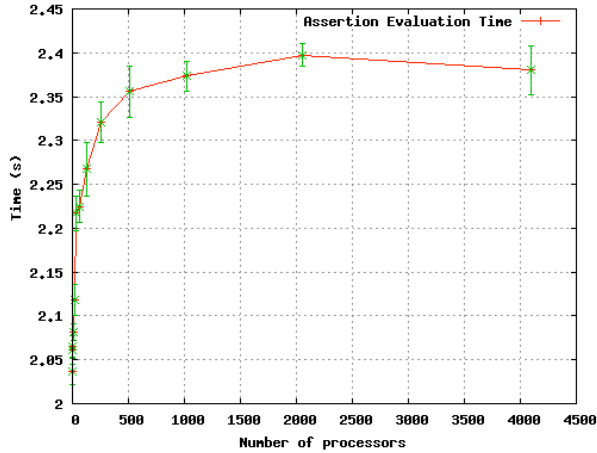


Figure 5 - Weak Scaling Assertion Evaluation Time

VI. CONCLUSIONS AND FUTURE WORK

Debugging parallel applications is becoming increasingly difficult as the number of processors increases and as scientific applications become more complex. The state of the art has not advanced significantly over the years. Today's parallel debuggers are little more than sequential ones that happen to handle multiple processes.

Our earlier work on relative debugging motivated the work in this paper. In particular, we have focused on techniques that are *data centric*, and can be used in the same way regardless of the number of independent threads of computation. We have reintroduced an old idea in programming language research, namely program invariants or assertions, as a debug time tool. We have illustrated that these can help locate errors without requiring a user to look at every value in a large data structure. Importantly, we have shown that in a number of cases, the parallel platform can be used to execute the logic of the assertion, making it efficient for large data structures and machines.

The work reported here is a proof of concept, and will be developed further. First, the current assertion syntax, whilst simple, might need to be extended and made more powerful. It uses a declarative notation with arrays as first class objects. However, a more powerful set of semantics might be required in order to support more complicated yet sophisticated debugging expressions to be injected into assertion statements. Second, no single debugging technique can address all problems. Thus, we envisage the integration of these ideas with other tools and approaches. To this end, we will leverage the Eclipse Parallel Tools Plugin (PTP) [21] project that is building a framework in which multiple tools can sit side-by-side in a powerful Integrated Development Environment (IDE). PTP supports efficient invocation of parallel programs from Eclipse environment. Therefore, it can help improving the process to invoke debug servers. Having previously integrated Guard into Eclipse PTP [12], we feel this will be an effective framework for further experiments in debugging.

ACKNOWLEDGMENT

This project is supported by the Australian Research Council under the Linkage grant scheme, and is supported by Cray Inc. We acknowledge our colleagues in the MeSSAGE Lab at Monash University. This material is based upon work supported by the Defence Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defence Advanced Research Projects Agency.

REFERENCES

- [1] NovoSpark Corporation, "NovoSpark Visualizer". Retrieved 10/04/2010, from <http://www.novospark.com/>
- [2] E. W. Dijkstra, *Structured programming*, 1969.
- [3] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, vol. 12, pp. 576 - 580, 1969.
- [4] B. Meyer, *Object-Oriented Software Construction*: Prentice Hall, 1997.
- [5] I. Dolphin Interconnect Solutions, "TotalView Debugger: A comprehensive debugging solution for demanding multi-core applications". Retrieved 03/04/2009, from <http://www.totalviewtech.com/pdf/TotalViewDebug.pdf>
- [6] F. Gioachin and L. V. Kal'e, "Dynamic High-Level Scripting in Parallel Applications", in *International Symposium on Parallel&Distributed Processing*, Roma, 2009, pp. 1-11.
- [7] G. R. Watson, "The Design and Implementation of a Parallel Relative Debugger", in *Faculty of Information Technology*. vol. PhD Thesis Melbourne: Monash University, 2000, p. 197.
- [8] H. Foster, A. Krotnik, and D. Lacey, *Assertion-Based Design*, 2 ed. Boston: Kluwer Academic Publishers, 2005.
- [9] Free Software Foundation Inc, "GDB: The GNU Project Debugger". Retrieved 15/01/2009, from <http://www.gnu.org/software/gdb/>
- [10] D. Cheng and R. Hood, "A Portable Debugger for Parallel and Distributed Programs", in *Conference on High Performance Networking and Computing*, Washington, D.C., United States 1994, pp. 723-732.
- [11] Allinea, "Allinea DDT - A revolution in debugging", 2009.
- [12] D. Abramson, T. Ho, C. Chu, and W. Goscinski, "Eclipse Guard: Relative Debugging in the Eclipse Framework". Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.128.5814&rep=rep1&type=pdf>
- [13] A. Petit, "Block Cyclic Data Distribution". Retrieved 20/12/2009, from <http://www.netlib.org/utk/papers/scalapack/node8.html>
- [14] H. Richardson, "High Performance Fortran: history, overview and current developments", Thinking Machines Corporation 1996. Retrieved from
- [15] D. Abramson, M. N. Dinh, D. Kurniawan, B. Moench, and L. DeRose, "Data Centric Highly Parallel Debugging", in *ACM International Symposium on High Performance Distributed Computing (HPDC)* Chicago, Illinois, 2010.
- [16] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-based Multicast/Reduction Network for Scalable Tools", in *Proceedings of SC*, Phoenix, AZ, 2003.
- [17] D. Abramson and G. Edgan, "The RMIT Data Flow Computer: A Hybrid Architecture", *The Computer Journal*, vol. 33, pp. 230-240, 1990.
- [18] D. A. Randall, "The Shallow Water Equations", Department of Atmospheric Science, Colorado State University, Fort Collins 2006. Retrieved from <http://kiwi.atmos.colostate.edu/group/dave/pdf/ShallowWater.pdf>

- [19] J. Michalakes, J. Hacker, R. Loft, M. O.McCracken, A. Snavely, and N.-I. J. Wright, "WRF Nature Run", in *High Performance Networking and Computing*, Nevada, 2007.
- [20] D. Abramson, M. Dix, and P. Whiting, "A Study of the Shallow Water Equations on Various Parallel Architectures", in *14th Australian Computer Science Conference*, Sydney, 1991, pp. 1-12.
- [21] Eclipse, "PTP - Parallel Tools Platform". Retrieved 18/07/2009, from <http://www.eclipse.org/ptp/>
- [22] S. F. Siegel and T. K. Zirkel, "Collective assertions", in *12th International Conference, VMCAI, Lecture Notes in Computer Science* vol. 6538, pp. 387-402, 2011.
- [23] A. Vo, S. Aananthakrishnan, and G. Gopalakrishnan, "A Scalable and Distributed Dynamic Formal Verifier for MPI Programs", in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.
- [24] Z. Chen, Q. Gao, W. Zhang, and F. Qin, "FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking". in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.