

# Relative Debugging for Parallel Systems

David Abramson and Greg Watson  
Department of Digital Systems,  
Monash University  
Clayton, VIC 3168

## Abstract

Relative debugging is a system which allows a programmer to compare the state of two executing programs. It is embedded in a conventional debugger, and makes it possible to isolate errors after a program has been modified or "ported" from one system to another. It has found been that relative debugging is useful for finding errors in large scientific programs. The technique is applicable to sequential programs, but is extremely valuable when sequential programs are ported to parallel platforms, because complex errors are often introduced at this stage.

Accordingly, we have been investigating the application of relative debugging to parallel programs. Whilst the general approach used for sequential programs can be applied, the implementation becomes much more difficult. We have developed a new execution mechanism based on data-flow semantics, which allows the debugger to control a number of processes. Further, we are currently building a prototype parallel debugger, and are exploring how a programmer might interact with such a tool. This includes issues in common with other parallel debuggers, but also requires research into appropriate mechanisms for specifying how comparisons are performed.

## 1.0 Introduction

Relative debugging is a system which allows a programmer to compare the state of two executing programs. It is embedded in a conventional debugger, and makes it possible to isolate errors after a program has been modified or "ported" from one system to another. We have previously demonstrated the applicability of relative debugging for finding errors in large scientific computations [1 - 6]. The technique is applicable to sequential programs, but is extremely valuable when sequential programs are ported to parallel platforms, because complex errors are often introduced at this stage.

A relative debugger needs to provide two main mechanisms for comparing the state of two programs, namely a *declarative* mode and an *imperative* mode. In the declarative mode, a programmer is able to assert that two data structures should be equivalent at particular locations in the program. This is done using a special command called `assert`, which contains the name of two data structures in two different programs, and also a line number in each program at which the data should be compared. The following assertion illustrates the key attributes:

```
assert P1::X@150 == P2::Y@80
```

Here, the data structure X in program P1 should be equal to the data structure Y in program P2, at lines 150 and 80 respectively. Declarative assertions are made a-priori, and are executed automatically by the debugger at the specified locations.

The imperative mode more closely resembles traditional debugger commands, in that it is executed as soon as it is issued. However, unlike conventional debugger commands, an imperative comparison takes data from two already suspended programs and compares the data for equality. It has the following form:

```
compare P1::X P2::Y
```

Both debugging mechanisms are useful for finding errors – Assertions are useful for detecting when an error has occurred, and imperative comparisons are useful for comparing data structures after a trigger event has occurred. For example, a divergence in one data structure may be sufficient to trigger an event, and then a more detailed set of comparisons can be performed imperatively.

In 1995 we built a relative debugger called Guard95. Guard95 incorporated a fairly simple mechanism for implementing assertions. In this scheme, assertions are parsed and the information about them is stored in an assertion list, which contains a list of pairs of symbol names and breakpoint addresses. In Guard95 a special statement called `verify` launches the two programs after placing breakpoints for each of the assertions in the assertion list. When breakpoints occur in both programs, Guard searches the assertion list for a matching pair, and subsequently retrieves the data of the named data structures. If they are equivalent, then the programs are restarted, but if they differ, control is returned to the user. This execution mechanism is powerful enough to handle the most common modes of interaction with sequential programs. However, it does not cater for cases in which a term (a data structure name and line number) appears in more than one assertion. It also is incapable of handling the complex control flow of parallel programs. In late 1996 we defined a new execution mechanism based on the data flow principle [3], and began the construction of a new debugger called Guard97. This paper discusses some of the design and implementation issues for Guard97. Much more information about relative debugging for sequential programs is available in a number of other publications [1 - 6].

The paper begins with a discussion about the major design issues for Guard97, and then illustrates the structure which has been adopted.

## **2.0 Design Issues**

### **2.1 Parallel Programming Methodology**

Guard97 incorporates mechanisms for supporting two major parallel programming methodologies, message passing and data parallel.

Message passing provides complete flexibility in the design of the parallel algorithm and the way in which data is distributed across the processes. For example, even simple data structures such as 2-D arrays can be distributed using a number of different methods, and in many cases individual processes may not know the size and shape of data until run-time (see §2.3). In order to cater for all these possible cases, a similar degree of flexibility must be provided by the debugger, either through a command language with programming constructs, or some other mechanism.

Data parallel languages generally place some restrictions on how data is decomposed, and hence reduce some of the flexibility issues discussed above. Because the mapping is performed by the data parallel run time system, the debugger must understand the way that data is distributed across processes. We have chosen to experiment with a prototype data parallel language called ZPL [10], developed at the University of Washington. ZPL uses a complex data structure at run time called an *ensemble* to describe the actual run time mapping. Accordingly, special knowledge must be included for each language that is to be supported – greatly increasing development and maintenance costs of the debugger.

### **2.2 Implementing Assertions using Dataflow**

As discussed, the original sequential execution mechanism implemented in Guard95 is not powerful enough to support assertions in a parallel environment. To fully support these assertions, the debugger needs to be able to:

- control the execution of multiple processes;
- extract data from individual processes in an independent manner;
- reuse this data in multiple assertions;

- receive and process events indicating changes in program execution status (e.g. reaching a breakpoint); and
- use the results of the data comparison to control the state of the debugger.

We have found that a dataflow execution mechanism [7, 8] provides the most suitable mechanism for meeting these requirements. Using this model, the assertion statement can be represented as a *graph* that compares data only when it has received input tokens (containing the data) from two (or more) processes. Since data can arrive from the processes at any time, the model provides a convenient mechanism for storing data until each node's inputs are satisfied and the assertion is fired.

Multiple assertions that share data from the same process location require the data to be duplicated for each assertion. In an assertion graph, edges are used to transmit tokens between the nodes. Where edges connect the output of one node to the input of multiple other nodes, this token duplication is managed automatically.

In the following example, S is a serial code, P is the corresponding 2 process parallel code. X and Y are equivalent data structures. Figure 2.0 shows the graph corresponding to the following assertions:

```
assert S::X[100..199]@150 == P[0]::Y@80
assert S::X[0..19]@100 == P[0]::Y[0..19]@50
assert S::X[0..19]@100 == P[1]::Y[0..19]@50
```

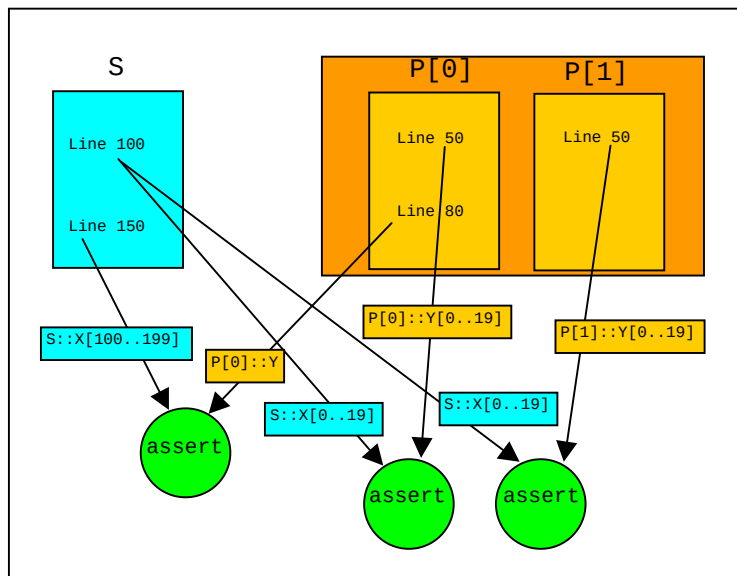


**Figure 2.0: Dataflow Example**

The first assert statement will not perform the comparison until the data X[100..199] at line 150 in S and Y at line 80 in P[0] have arrived on both inputs. The two remaining asserts show that the data X[0..19] at line 100 in S is automatically duplicated and used in comparisons with Y[0..19] at line 50 in P[0] and Y[0..19] at line 50 in P[1]. The use of data flow graphs is discussed more completely in [3].

### 2.3 Dynamic Assertions

In many parallel codes, the details of the data decomposition and the number of processes involved in the computation may not be known until run time. Simple static assertions as discussed in the previous section do not provide enough flexibility to deal with these cases.

To handle such dynamic problems, additional functionality needs to be provided to allow the necessary information to be *extracted* from the running executable prior to the assertion being applied. Assertions that contain these initially undefined quantities are known as **dynamic assertions**.

For example, some parallel programs send the actual bounds of distributed data structures to the processes only after they have been computed at run time. This means that individual processes are not aware of which section of data they are computing until they begin execution. Figure 2.1 shows such a situation. The vector  $X$  in the serial code contains  $N$  elements. In the corresponding parallel code, each of  $P$  processes would compute a portion of the vector corresponding to  $Y[start..end]$ , where  $(end - start) * P = N$ , and  $end$  and  $start$  are stored in each process but not initialised until run time (perhaps from a master process). The corresponding dynamic assertion in this case (for a fixed number of processes  $P$ , and assuming processes are numbered from 0 to  $P-1$ ) would be:

```
assert s::X[p[i]::start@20..p[i]::end@20]@30 == p[i]::X[0..N/P]@50  $\forall i \in \{0..P-1\}$ 
```

## 2.4 Process Management

Many parallel message passing standards like MPI do not actually define a mechanism for process initialisation. Instead, they rely on the individual implementations to provide a script, or program, that handles the initial process creation and ensures that the appropriate information is passed to the processes.

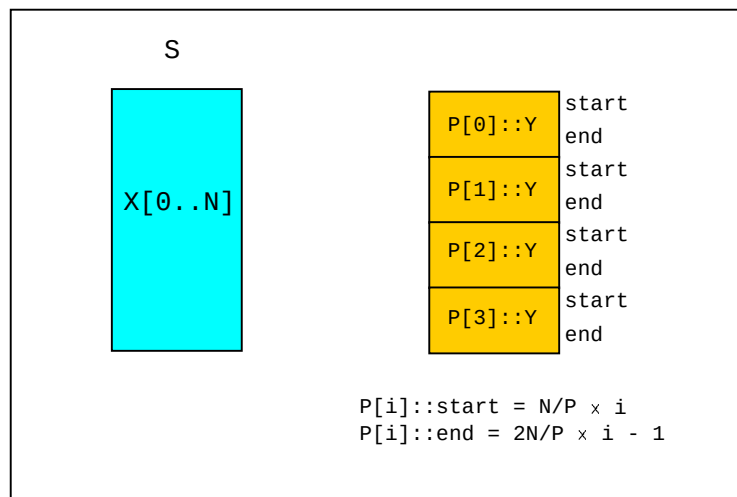


**Figure 2.1: Dynamic Assertions Example**

Unfortunately, no specific debugging support is built into the standard. Accordingly, a debugger must resort to different methods for each message passing implementation in order to attach to and control the individual processes.

Currently, Guard97 only supports static process creation through a debugger `invoke` statement. In this scheme, the user specifies the number of processes that are to be created statically, then refers to individual processes in `assert` statements by indexing the process name with a process task id. This is consistent with the process creation mechanisms provided by IBM's Parallel Operating Environment (POE) and the Message Passing Interface (MPI).

As mentioned previously, some parallel codes may not allocate all processes statically, but rather create and destroy processes under the control of the application. In particular, the message passing standards PVM [9] and MPI-2 [11] provide this capability. However, such dynamic process creation raises a number of issues including:

- notification of new processes creation;
- finding the current number of processes;
- referring to new processes; and
- attaching to and controlling new processes.

Techniques for supporting dynamic process creation are currently under development.

### 3.0 Implementation Issues

#### 3.1 Interactive User Interface

Like Guard95, Guard97 uses a simple command oriented interactive interface. There are two reasons for this. First, most available debuggers are command line driven, so users are very familiar with this style of interface. Second, graphical user interfaces can be relatively easily added at a later stage, but generally require significantly more development effort.

The command set available is a subset of that provided by GNU's GDB [13]. Each command can also be prefixed by a process name, which can be a simple name identifying a serial process or group of parallel processes, or indexed by a process task id to identify an individual parallel process. If no process is specified, the command will operate on the current process. The special process identifier *all:* can be used to identify all processes currently under the control of the debugger.

In addition, the following commands have been added to support relative debugging:

*invoke process-name*{*[num-procs]*} "*command*" {using "*mpich* | *poe*"}

Invoke the process *process-name* specified by *command*, optionally as a parallel process of *num-procs* processes and using the message passing architecture *mpich* or *POE*.

*assert p1::variable1@line1 == p2::variable2@line2*

Define an assertion that *variable1* at *line1* in process *p1* will be equivalent to *variable2* at *line2* in process *p2*.

*set error lower upper relative|absolute*

Set the error threshold for the following assert statements. If  $\delta$  is the difference value in each assert, then the graph will continue executing while  $\delta < upper$ . Difference values will be displayed while  $lower \leq \delta \leq upper$ .

These commands can be compiled into a dataflow graph, which is then started using a special **START** command. Typically, a graph will contain one **invoke** statement for each program, plus one or more assertions.

#### 3.2 Client/Server Architecture

In order to operate in a parallel environment, Guard97 must be able to control processes executing on various CPU's possibly distributed across a network. The current implementation does this with a client/server architecture using the Sun Microsystems RPC communication mechanism [14]. Figure 3.0 shows Guard97 in operation.

The client consists of the command line user interface, a graph compiler, and the dataflow interpreter. Interactive commands are converted into remote procedure calls, which are passed to the servers associated with the process name specified in the command (or the default process). Assertions and program invoke

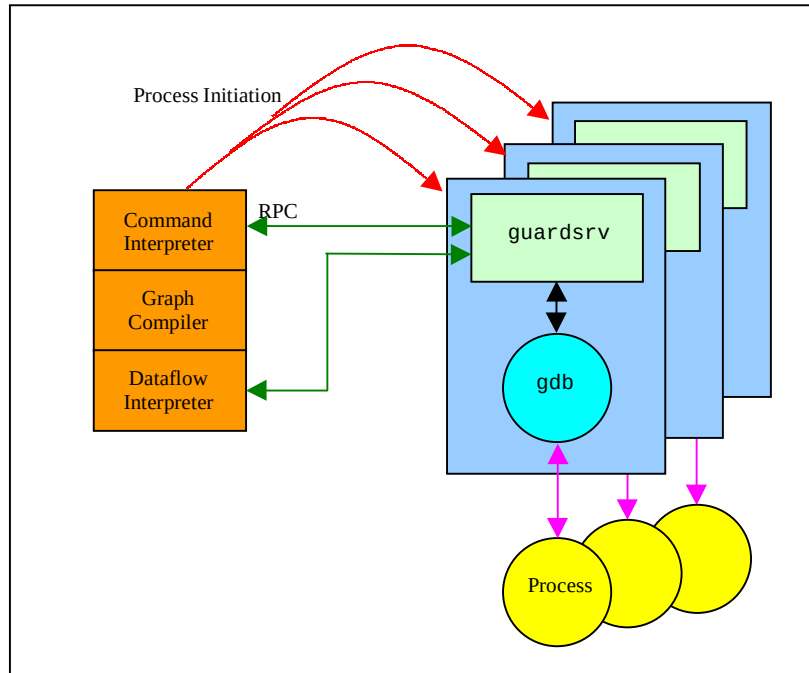


Figure 3.0: Operation of Guard97

statements are passed to a graph compiler which generates a dataflow graph. A user can execute a graph using the `start` command, which passes the compiled graph to the dataflow interpreter. The interpreter will then in turn, issue RPC calls to the servers as necessary. The graph will then invoke the programs, plant the breakpoints and process the breakpoint interrupts as they occur.

When each (parallel or serial) process is to be executed on a CPU, Guard97 first starts a debug server (`guardsrv`) on the CPU, hence there is one instance of `guardsrv` for each process controlled by Guard97. The debug server begins by starting the process under debugger control, then simply waits for remote procedure calls from the client. The server decodes the calls and performs the appropriate debugger operation on the process. The result of the operation is then returned to the client.

### 3.3 GDB

In order to avoid the cost of developing and maintaining the low level debugging functions employed by the debug servers, we have chosen to use GNU's GDB as a back-end debugger. There are significant advantages in adopting this approach, including:

- GDB is a stable widely used debugger;
- GDB's interface is relatively static;
- source code to GDB is widely available;
- GDB has been ported to over 20 different architectures;
- additional functionality can be added easily.

When a Guard97 debug server (`guardsrv`) is initiated, it first starts GDB, then instructs GDB to load the actual process executable. The server can then control the process by sending commands to GDB, and interpreting GDB's responses to those commands. Using this mechanism, the server can respond to RPC messages that it receives from the client.

One problem with using GDB as the back-end debugger is the speed at which data can be extracted from the executing process through GDB's command line interface. This becomes a significant bottleneck when

large amounts of data are required. In order to overcome this performance limitation, we have provided an additional command to GDB to extract data in binary format. While not strictly necessary, this new command does improve the debugger performance markedly.

### **3.4 Architecture Independent Format**

Guard97 supports the debugging of codes on many architectures with different data representations. The debugger front-end must perform comparisons on data that has been obtained from these different architectures, but itself can be located on a different architecture from any of those executing processes under its control. In order to overcome problems with managing different data representations, Guard97 uses an architecture independent data format (AIF) when dealing with data from each process.

AIF supports simple types: characters, integers, and IEEE 754 floating point numbers. In addition all the commonly used complex data types are supported. Data is converted between AIF and the native format by appropriate conversion routines. A range of arithmetic and other operations is available to manipulate the data once it has been converted to AIF.

Internally, data in AIF consists of two components: a variable length format descriptor (FDS), and a variable length octet string containing the data. The FDS completely describes the type of the original data so that it may be reconstructed if necessary. Currently, only 32 & 64 bit integers and IEEE 754 floating point numbers are supported.

### **Conclusions**

A prototype version of Guard97 has been tested with some simple parallel message passing programs under MPI on a network of Sun Workstations and POE on an IBM SP2. It has demonstrated that the basic dataflow execution mechanism works.

Currently, the system only supports static assertions, and dataflow graphs must be hand coded in an assembler like notation. We are currently implementing dynamic assertions which will make it possible to model some common data decomposition strategies. We are also designing a high level programming language for dataflow graphs. This will make it possible to mix imperative debugger statements with assertions, and build complex graphs for debugging applications which use unconventional control flow and data decomposition techniques.

### **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. , Sosic, R., and Watson, G. "Implementation Techniques for a Parallel Relative Debugger", Proceedings of PACT'96, pp 218 – 226, October 1996.
- [4] 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.
- [5] 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

- [6] Abramson, D.A. and Sasic, R. "Relative Debugging using Multiple Program Versions", Key Note Address, 8th Int. Symp. on Languages for Intensional Programming , Sydney, 3-5th May, 1995. In Intensional Programming I, World Scientific, ISBN 981 - 02 - 2400 - 1.
- [7] Arvind, Bic, L. and Ungerer, T. "Evolution of Data-flow Computers", Chapter 1, "Advanced Topics in Dataflow Computing", Prentice Hall, 1991.
- [8] Dennis, J. "The Evolution of "Static" Data-flow Architectures", Chapter 2, "Advanced Topics in Dataflow Computing", Prentice Hall, 1991.
- [9] Dongarra, J., Geist, G., Manchek, R., and Sunderam, V. "Integrated PVM Framework Supports Heterogeneous Network Computing", Computers in Physics, April 1993.
- [10] Lin, C. and Snyder, L. "ZPL: An Array Sublanguage", Languages and Compilers for Parallel Computing, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds, pp. 96-114, 1993.
- [11] MPI Forum, "MPI-2: Extensions to the Message-Passing Interface", University of Tennessee, 1997.
- [12] Sasic, R. and Abramson, D. A. "Guard: A Relative Debugger", to appear, Software Practice and Experience.
- [13] Stallman, R. "Debugging with GDB – The GNU Source Level Debugger", Edition 4.12, Free Software Foundation, January 1994.
- [14] Sun Microsystems, Inc., "Remote Procedure Call Specification", RFC-1094, DDN Network Information Center, SRI International, Menlo Park, CA.