

Parallel Relative Debugging for Distributed Memory Applications: A Case Study

Gregory R. Watson and David Abramson

School of Computer Science and Software Engineering
Monash University
Wellington Road, Clayton, VIC 3168, AUSTRALIA

Abstract

Relative debugging is a technique that addresses the problem of debugging programs that have been developed using evolutionary software techniques. Recent developments allow relative debugging to be used on programs that have been ported from serial to parallel architectures or between different parallel architectures. Such programs may change significantly in the porting process and this raises many issues for the debugging methodology. This paper examines the use of relative debugging on a distributed memory application in which errors were introduced when the code was ported from a sequential to a parallel architecture. Our debugger, GUARD-2000, is used to compare key data structures between the two codes even though the parallel data structure has undergone significant reorganisation when mapped onto a distributed memory platform. We show how this technique can quickly and accurately pinpoint the source of errors in the parallel code.

INDEX TERMS: parallel programming tools, relative debugging, distributed memory, porting.

1 Introduction

Relative debugging is a technique that allows a user to compare data between two executing programs [17]. It was devised to aid the testing and debugging of programs that are either modified in some way or are ported to other computer platforms. Traditional debuggers force the programmer to understand the expected state and internal operation of a program. In contrast relative debugging makes it possible to trace errors by comparing the contents of data structures between programs at run time. In this

way the programmer is less concerned with the actual state of the program and more concerned with finding when and where differences between the old and new codes occur. Relative debugging operates by allowing the user to define a series of *assertions* between a *reference* program and a *suspect* program. These assertions specify key data structures that must be equivalent at specific locations in the codes. When the programs are executed the debugger is able to use the assertion information to automatically compare the data structures and report any differences to the user.

A number of case studies over the years have demonstrated the significant power of relative debugging as a practical technique for locating errors experimentally [5][18]. However, parallel programs provide additional challenges in debugging [8]. Importantly, in many cases they are converted from existing sequential codes rather than being rewritten from scratch. We have argued that relative debugging is an important tool aimed at simplifying the porting of sequential programs to parallel platforms because it recognises the existence of a working sequential version that can be used as a comparison point. Until recently we have only been able to demonstrate this for sequential programs and a restricted set of parallel codes. This was because our early implementations of the debugger did not have sufficient expressive power to deal with distribution of data across a parallel machine, nor the changes that are necessary in data structure when the program is ported.

In this paper we present a case study that illustrates the application of relative debugging *when a program is ported from a sequential language to a parallel version running on a distributed memory platform*. The example illustrates a novel technique that is required in such an environment, namely the ability to

describe the way the data is distributed across the memory of the parallel machine. The paper gives a very brief description of relative debugging, and its implementation called GUARD. It then describes the application under consideration which is known as the Shallow Water Equations [16]. This is followed by a discussion of how GUARD facilitates data comparison across distributed memory machines.

2 Relative Debugging

Unlike other conventional parallel debuggers [6][12][13] a relative debugger provides the ability to dynamically compare data between two executing programs regardless of their location and configuration. In our implementation of a relative debugger, called GUARD, data comparisons can be performed either using an *imperative* scheme or a *declarative* scheme. Imperative comparisons can be performed explicitly by the user when two programs under the control of the debugger are stopped at breakpoints. The imperative `compare` command behaves like a conventional debugger data display command, however it names data structures in two processes instead of one. For example the following command compares the data from array A in `$proc1` with array B in `$proc2`:

```
compare $proc1::A = $proc2::B
```

If the arrays differ then the differences are reported. The `compare` command requires a fairly high degree of interaction by the user because the two programs must be manually executed and halted before the data can be compared.

An alternative way of comparing data is to use the declarative `assert` command. This command takes a pair of data structure names, process and file names and line numbers and compares the contents of the data structures only when the processes have reached their respective line numbers. The following command compares the data from array A in `$proc1` with array B in `$proc2` when `$proc1` reaches line 123 and `$proc2` reaches line 456:

```
assert $proc1::A@"file1.c":123 =  
      $proc2::B@"file2.f":456
```

When the user issues an `assert` command the debugger compiles it into a dataflow graph

which is then executed under the control of an interpreter. Dataflow nodes only fire when the programs reach breakpoints and the comparison is performed only when both data structures are available.

The relative debugging methodology involves placing assertions at key points in the two programs until a difference in data is detected. At this point the user examines the control and data flow of the programs and new assertions are written with the aim of refining the region that is in error. This process means that it is possible to reduce the region under consideration to a small code fragment very quickly because the region is divided on each pass. Conventional debug methodology and tools can be used to finally isolate the error. It is beyond the scope of this paper to describe in detail how relative debugging is implemented however the scheme is fully described in [3]. We refer the reader to a number of other papers that provide a thorough introduction to relative debugging and how the technique can be used to find the location of errors [2][4].

The `assert` and `compare` commands provide a conceptually simple mechanism for comparing data between two sequential programs. However a key goal of relative debugging is to extend this mechanism to encompass the comparison of data between related sequential and parallel programs. In this situation the comparison process is complicated by the decomposition that is normally applied to sequential data structures when a code is ported to a parallel architecture. To address this issue GUARD provides a `map` function which is used to define the data decomposition that has been employed. Rather than just supporting standard decompositions such as `BLOCK` and `CYCLIC` [7] as found in languages like HPF [10] we allow the user to write an arbitrary arithmetic expression that dictates the conversion of indexes from one space to another. Once the mapping has been specified it can be used in an `assert` command to provide the debugger with the necessary information needed to perform the comparison operation. For example if the array B above has been distributed using a block-cyclic decomposition defined by the `map bcyc` then the assertion might be written as:

```
assert $proc1::A@"file1.c":123 =  
      bcyc($proc2::B@"file2.f":456)
```

Once a data distribution mapping has been defined the user no longer needs to be concerned with the decomposition issues and can again concentrate on locating the source of the error. A complete description of parallel relative debugging is given in [19]. The utility of this technique is demonstrated in the following case study which examines the use of relative debugging to locate errors in a distributed memory parallel code.

3 Distributed Memory Case Study

The program used in this case study is a simple numerical model of the shallow water equations [15][9]. Abramson, et al. ported the shallow water equations to a number of languages and architectures [1]. Both the sequential and parallel versions are written in the C language. The parallel program is a hand-coded distributed memory implementation using the MPI parallel architecture [14]. The application is interesting because it represents a simplified model of the dynamics of a real weather model [11] and it captures features of the computation such as the communication and data distribution requirements. The equations are discretized across a regular grid and a finite difference formulation is used to approximate the partial derivatives. The program time steps from one iteration to another updating the key variables on each step. Figure 1 shows the basic program call tree.

```

main()
{
    int ncy;

    initialise(u,v,p,psi,di,dj);
    ...
    for (ncy = 0; ncy < itmax; ncy++) {
        /* Calculate cu, cv, z and h */
        calcuvzh(p,u,v,cu,cv,z,h,fsdx,fsdy);
        ...
        /* Calculate time tendencies */
        timetend(dudt,dvdt,dpdt,z,cv,cu,h);
        ...
        /* Calculate new values */
        timestep(u,v,p,dudt,dvdt,dpdt,first,tdt);
        ...
        if ( first ) {
            /* leapfrog */
            tdt = tdt+tdt;
            first = 0;
        }
    } /* End of time step loop */
    ...
}

```

Figure 1: “shallow” program call tree

The distributed memory code uses a master/slave arrangement, where the master maintains primary copies of the key data structures. Each slave is sent an entire copy of the data structures, but only performs computations on a portion determined by slicing the outer loop using a traditional BLOCK decomposition [7]. Before and after a calculation, each slave synchronises the edges of the data structures with its immediate neighbours. At the end of the time step loop the slave data is copied back to the master process. This arrangement is shown in Figure 2.

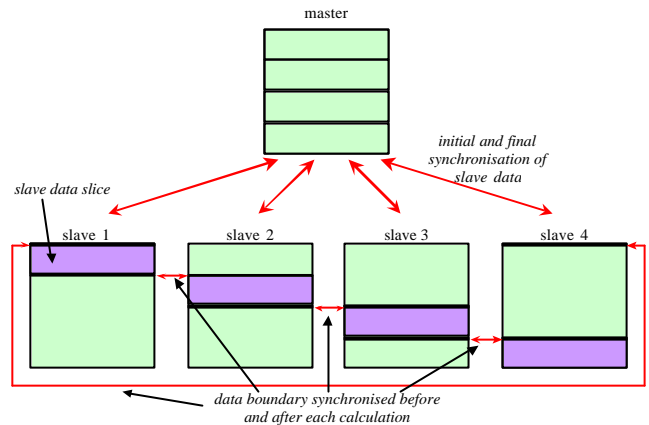


Figure 2: Data decomposition and boundary synchronisation for “shallow”

Both codes use a number of key variables in the computation of the shallow water equations. These include the zonal wind (u), meridional wind (v), pressure (p), potential enstrophy (z) and quantity (h) as well as mass weighted zonal wind (cu) and meridional wind (cv). In addition the codes maintain time tendency variables for the zonal wind ($dudt$), meridional wind ($dvdt$) and pressure ($dpdt$). Both codes compute new values for cu , cv , z and h in the routine `calcuvzh()` and perform the time tendency calculations in the routine `timetend()`.

In order to begin debugging the “shallow” code, a map that describes the data decomposition technique used by the distributed memory code was defined. One major difference between the codes is the index ordering which has been swapped in the translation from sequential to parallel. The map was also defined to account for this index permutation.

```

map shallow(P:A)
  define index(i,a,u,l) = a[l-i]
  define proc(i,a,u,l) =
    a[l] * $procs / $rows
end

```

The first line of the map indicates that the i^{th} index of the array should be swapped with the $(1-i)^{\text{th}}$ index. This effectively swaps the indices of a two-dimensional array. The second line of the map specifies which process holds each element of the array. In this case multiplying the first index value by the number of processes and dividing by the number of rows in the array determines the process number. i.e. rows 0 through $\text{rows}/\text{procs}-1$ are located on process 0, rows/procs through $2*\text{rows}/\text{procs}-1$ are located on process 1, and so on. The full mapping syntax and semantics can be found in [19].

For this case study a model size of 32×32 and a run length of 950 time steps was chosen. The distributed memory code was initially run on two processors. The codes are designed to report a number of key values every 50 time steps so it was simple to determine if the codes were working correctly or not. Figures 3a and 3b show the output from the two versions of the code after 950 time steps and in which significant errors can be clearly seen in all four values.

```

Cycle number 950
Model time in days 0.99
Potential energy 6816.106
Kinetic Energy 41183.156
Pot. Enstrophy 2.006166e-27
Total Energy 47999.262

```

(a) Sequential

```

Cycle number 950
Model time in days 0.99
Potential energy 6790.644
Kinetic Energy 41219.078
Pot. Enstrophy 6.152948e-17
Total Energy 48009.723

```

(b) Distributed Memory

Figure 3: Output from the “shallow” code

3.1 Error 1: Incorrect Index Value

The shallow water mapping was used to define a series of assertions to compare the values of the key data structures u , v and p between the sequential and parallel codes just

prior to entering the main time step loop. This tests the initialisation code in the two programs, and ensures that the variables are correct before any calculations are undertaken. The error limits were set to an initial estimate of the likely error range. The commands used were as follows (assuming $\$c$ represents the sequential program and $\$p$ the parallel program):

```

set error 1.0e-20 1.0e-1
assert $c::u@"cshallow.c":61 =
  shallow($p::u@"main.c":405)
assert $c::p@"cshallow.c":61 =
  shallow($p::p@"main.c":405)
assert $c::v@"cshallow.c":61 =
  shallow($p::v@"main.c":405)

```

Execution of these assertions showed that u , p and v are correct prior to entering the time step loop. This confirms that the errors must be being introduced by calculations in this loop.

The next step was to examine the result of the computations performed in `calcvzh()`. This was done by defining assertions to compare the values of the variables cu , cv , h and z as follows:

```

assert $c::cu@"cshallow.c":71 =
  shallow($p::cu@"main.c":430)
assert $c::cv@"cshallow.c":71 =
  shallow($p::cv@"main.c":430)
assert $c::h@"cshallow.c":71 =
  shallow($p::h@"main.c":430)
assert $c::z@"cshallow.c":71 =
  shallow($p::z@"main.c":430)

```

After first call to `calcvzh()` differences were observed in the variable cv . These differences are shown in Figure 4a. To visualise the differences in the arrays we use a two-dimensional representation where each pixel shows the value of the difference at the corresponding location in the arrays. The relative magnitude of the differences are mapped onto a colour table with white representing no differences and black the largest difference. Using this technique the differences observed in the cv arrays suggested that the error was related to a boundary calculation by each processor.

To confirm this suspicion the codes were re-run with the distributed memory code executing on four processors rather than the initial two. Errors were again observed in the variable cv however this time the differences appeared as shown in Figure 4b. This confirmed that cv was affected by a boundary issue that depended on number of processes. Since the error was in the

boundary column the corresponding boundary code was examined for coding defects. Inspection of the `calcuvzh()` code (12 lines) lead to the discovery of an incorrect array index in the calculation of `cv`.

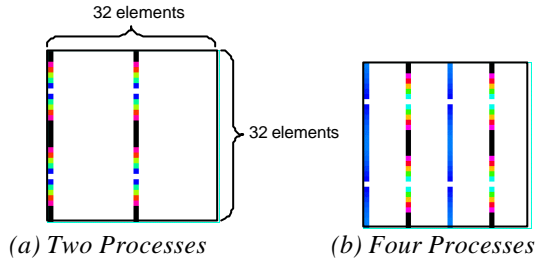


Figure 4: Boundary errors observed in the `cv` variable

Relative debugging allowed the rapid identification of this error by observing the changes resulting from running on different numbers of processors, and irrespective of the fact that the array indexes are permuted.

3.2 Error 2: Wrong Array Element

After correcting the index value and re-running the programs using the same assertions errors were still observed in the `cv` variable. The characteristics of the differences had altered significantly however and now seemed to indicate a periodic error in the `cv` calculation. The new differences are shown in Figure 5.

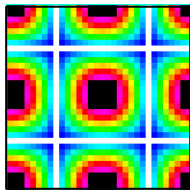


Figure 5: Apparent periodic errors in the `cv` variable

Further inspection of the `calcuvzh()` revealed a second error in the calculation of `cv`. In this case the wrong element of the array `v` was used in the calculation of `cv`.

This error was interesting because it occurred in the same statement as the previous error, but was missed in the original code inspection. Because the nature of the error in a single variable changed from a boundary problem to a periodic problem the source of the problem was quickly identified.

3.3 Error 3: Wrong Sign

After ensuring that both errors had been corrected in the parallel code the programs were rerun to check that the `cv` variable was now correct. The assertions showed that all four calculated variables, `cu`, `cv`, `h` and `z` were correct after first iteration, however on subsequent iterations errors were observed accumulating in the `cu` variable. Figure 6 shows an example of these errors.

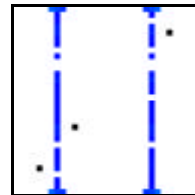


Figure 6: Errors observed accumulating in the `cu` variable after first time step

Inspection of the code showed that the calculation of `cu` is derived from the values of `p` and `u`. Since all values of `cu` are correct on first iteration but incorrect on second and subsequent iterations, this implies that there must be in error in the computation of `p` and `u` (which were shown to be correct immediately after initialisation).

The following assertions were defined in order to check the values of `p` and `u` at the end of the time step loop:

```
assert $c::u@"cshallow.c":85 =
    shallow($p::u@"main.c":449)
assert $c::p@"cshallow.c":85 =
    shallow($p::p@"main.c":449)
```

When the codes were again run errors were immediately reported in `u`. These are shown in Figure 7.

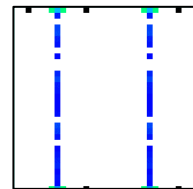


Figure 7: Errors observed in `u`, which is used to derive `cu`

At this point it was clear that errors were being introduced into the variable `u` but these could have resulted from either the calculation of

u itself in `tstep()` or in the time tendency variable `dudt` which is calculated in `timetend()`. To verify which routine was the source of the errors it was necessary to rerun and check the value of `dudt` immediately after the call to `timetend()`. Figure 8 shows the differences visible in `dudt` when this was done.

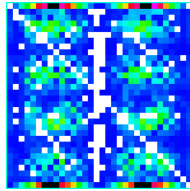


Figure 8: Errors resulting from wrong sign in calculation of `dudt`

It was now known with reasonable certainty that the errors were occurring in the `timetend()` routine. At this point a visual inspection of the 15 lines of code in `timetend()` was used to locate the source of the error. In this case the error turned out to be a wrong sign used in the calculation of `dudt`.

This error highlights the use of relative debugging to quickly reduce the possible location of an error to a small region of code. Once this has been done, simple code inspection will usually pinpoint the incorrect statement immediately even when the actual error is very minor as in this case.

4 Conclusion

In this paper we show how our relative debugger, GUARD, can be used to locate errors in a code that has been ported from a sequential architecture to a distributed memory architecture. This work is significant because it is the first time that relative debugging has been used to locate errors in a hand-coded distributed memory code running on multiple processes. The case study shows that the debugger:

- easily handles significant changes to key data structures that are introduced during the porting process; and
- can accurately locate the source of errors in only a few steps using an iterative refinement process.

One criticism of the mechanism used by the debugger to describe data decomposition is that the user may write an incorrect transformation and thus the debugger will indicate that the

structures are incorrect even though they are actually the same. In addition the mapping language itself is inherently fragile. That is, minor coding errors can lead to major changes in the final specification. The consequences of such mapping errors are that it may become difficult or impossible to locate the original program errors or that ‘spurious’ errors may be introduced. We propose a number of solutions to overcome these shortcomings. First we believe that the use of pre-packaged transformations will reduce the occurrence of this situation. It should be possible to provide transformations to replicate most normal decomposition techniques such as block and block-cyclic. Map creation could also be facilitated through the use of either a semi- or fully automated analysis system. A semi-automated system could provide a graphical environment for specifying maps and allowing the user to model and test the decomposition or transformation interactively. A fully automated system could perform code analysis on the serial and parallel codes to determine the required mapping.

In conclusion it is worth indicating that relative debugging is of limited value when timing and other non-deterministic errors occur in parallel codes. The technique is very effective at locating a region that is in error and it will flag errors between a working program and an erroneous one. However, because the debugger disturbs the timing of the program any errors it isolates may vary from one execution to another if non-deterministic behaviour is present.

Acknowledgements

This work has been supported by the Australian Research Council under the Large Grant Scheme and by Microsoft Corporation.

References

- [1] Abramson, D., Dix, M. and Whiting, P., “A Study of the Shallow Water Equations on Various Parallel Architectures”, *14th Australian Computer Science Conference*, pp. 06-1 – 06-12, Sydney, 1991.
- [2] Abramson, D., Sobic, R., “A Debugging Tool for Software Evolution”, *CASE-95, 7th International Workshop on Computer-Aided Software Engineering*, pp 206 – 214, Toronto, Canada, July 1995.

- [3] Abramson, D., Susic, R., Watson, G., "Implementation Techniques for a Parallel Relative Debugger", *Proceedings of PACT '96*, Boston, October 1996.
- [4] Abramson, D., Foster, I., Michalakes, J., Susic, R., "Relative Debugging: A New Methodology for Debugging Scientific Applications", *Communications of the ACM*, Vol. 39, No. 11, pp 69 – 77, November 1996.
- [5] Abramson D., Foster, I., Michalakes, J., Susic R., "Relative Debugging and its Application to the Development of Large Numerical Models", *Proceedings of IEEE Supercomputing 1995*, San Diego, December 1995.
- [6] Cheng, D., Hood, R., "A Portable Debugger for Parallel and Distributed Programs", *Proceedings of Supercomputing '94*, pp. 723 – 732, November 1994.
- [7] Foster, I., *Designing and Building Parallel Program: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley Publishing Co., 1995.
- [8] Gait, J., "A Probe Effect in Concurrent Programs", *Software Practice and Experience*, Vol. 16, No. 3, pp. 225 – 233, 1986.
- [9] Haltiner, G.J. and Williams, R.T., "Numerical Prediction and Dynamic Meteorology", Wiley, 1980.
- [10] High Performance Fortran Forum, "High Performance Fortran Language Specification", *CRPC-TR92225*, Version 2.0, Center for Research on Parallel Computation, Rice University, , January 1997.
- [11] Hoffmann, G.-R. and Maretis, D.K., *The Dawn of Massively Parallel Multiprocessing in Meteorology*, Springer-Verlag, 1990.
- [12] Kacsuck, P., Cunha, J., Dózsa, G., Joureço, J., "A Graphical Development and Debugging Environment for Parallel Programs", *Parallel Computing*, Vol. 22, No. 13, pp. 1747 – 1770, 1997.
- [13] May, J., Berman, F., "Panorama: A Portable Extensible Parallel Debugger", *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 96 – 106, San Diego, May 1993.
- [14] Message Passing Interface Forum, *MPI: A message-passing interface standard*, International Journal of Supercomputer Applications, Vol. 8, No. 3/4, pp 165 – 414, 1994.
- [15] Pedlosky, J., *Geophysical Fluid Dynamics*, Springer-Verlag, 1979.
- [16] Sadourny, R., "The Dynamics of Finite-Difference Models of the Shallow Water Equations", *Journal of Atmospheric Science*, Vol. 32, pp. 680 – 689, 1975.
- [17] Susic, R., Abramson, D., "Guard: A Relative Debugger", *Software – Practice and Experience*, Vol. 27, No. 2, pp 185 – 206, February 1997.
- [18] Watson, G. and Abramson, D. "Relative Debugging For Data Parallel Programs: A ZPL Case Study", *IEEE Concurrency*, Vol 8, No 4, pp 42 – 52, October 2000.
- [19] Watson, G. R., *The Design and Implementation of a Parallel Relative Debugger*, PhD Thesis, Monash University, Melbourne, October 2000.