

# Relative Debugging: A new methodology for debugging scientific applications

David Abramson §

Ian Foster †

John Michalakes †

Rok Sosić §

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

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

## Abstract

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

## 1. Introduction

Large scientific codes are constantly evolving. Refinements in understanding of physical phenomena result in changes to physics, improved numerical methods result in changes to solution techniques, and developments in computer architecture result in new algorithms. Unfortunately, this evolutionary process often introduces subtle errors that can be extremely difficult to find. As a consequence, scientific programmers can spend many hours, days, or weeks laboriously comparing the executions of two almost identical codes, seeking to identify the source of a small discrepancy.

Debuggers assist in locating program errors. They are tools which allow a user to investigate the execution state of an application program, by for example examining the state of program variables. Significant recent research includes the addition of graphical user interfaces to improve the ease of use, data visualization facilities to aid the interpretation of large and complex data structures, the concept of process groups to aid the management of many independent threads in parallel machines, and support for parallel and distributed debugging [5, 6, 7, 8].

Traditional debuggers have proved invaluable when developing new programs. However, they do not directly address the problems of maintaining and extending

existing computer programs, or converting software from one machine or language to another. Programmers do not want to examine new versions of existing programs in isolation. Instead, they want to *compare* their execution with the execution of an old, reference program which is assumed to be correct. By acting as a reference, the working version can assist in locating the section of code in the modified program which introduces incorrect values.

Existing techniques for comparing executions of two program versions are tedious, error prone and limited in scope. For example, a programmer may invoke the two programs under separate debuggers, manually set breakpoints, run the programs, and visually compare the resulting program states. A more advanced approach is to insert output statements into both programs and then compare the output using a file comparison program [4]. This approach also has its limitations: it can require huge amounts of disk storage for the program output, involves modifications to both programs, and is not easily extended to take into account data types. For example, if floating point numbers are being compared, then programs that compare files character by character may not be sufficiently flexible.

In this paper we describe a new debugging methodology called *relative debugging* that addresses these difficulties. The key idea in relative debugging is that errors in a new version of a program can be located by automated, runtime comparison of the internal state of the new and reference versions. When supported by appropriate tools, this approach does not require any modifications to user programs and can perform comparisons on the fly, without requiring disk storage. Comparisons can take into account differences in data representations of the two programs, making it possible for the new and reference versions to run on different machines or to be written in different languages. The latter feature is particularly important in supercomputing applications, for example when porting a vector code to a parallel computer.

In addition to introducing relative debugging, this paper describes a particular instantiation of this methodology - the relative debugger Guard - and presents the results of an experimental study in which Guard was applied to a large scientific code. Guard uses a machine independent debugging interface to support relative debugging in heterogeneous, multilanguage environments. The experimental study involves a mesoscale atmospheric circulation model called MM5, and demonstrates the power of Guard by using an existing sequential version to account for subtle numeric differences in a parallel version.

## **2. Relative Debugging**

Traditional debuggers allow a user to control a program and examine its state at any point of the execution. The user sets breakpoints in the code, interactively examines program variables, and verifies that these variables have expected values. Erroneous values can be traced to erroneous code by using information about program data flow. Relative debugging differs from traditional debugging in two important respects. First, program variables are compared not with user expectations, but with variables in another reference program that is known to be correct. Second, because the reference program is available to compute correct values, the comparison process can be automated. Hence, the relative debugging process proceeds as follows. The user first formulates a set of assertions about key data structures in the reference and the development versions. These assertions specify locations at which data structures should be identical: violations of the assertions indicate errors. The relative debugger is then responsible for managing the execution of the two program versions, for validating the supplied assertions by comparing the data structures, and for reporting any differences. If differences are reported, the user proceeds to isolate erroneous code by repeatedly refining the assertions and reinvoking the relative debugger. Once the erroneous region is small enough, traditional debugging techniques can be used to

correct the development version. Thus, the relative debugger provides a quick and effective way of locating problems in the development version.

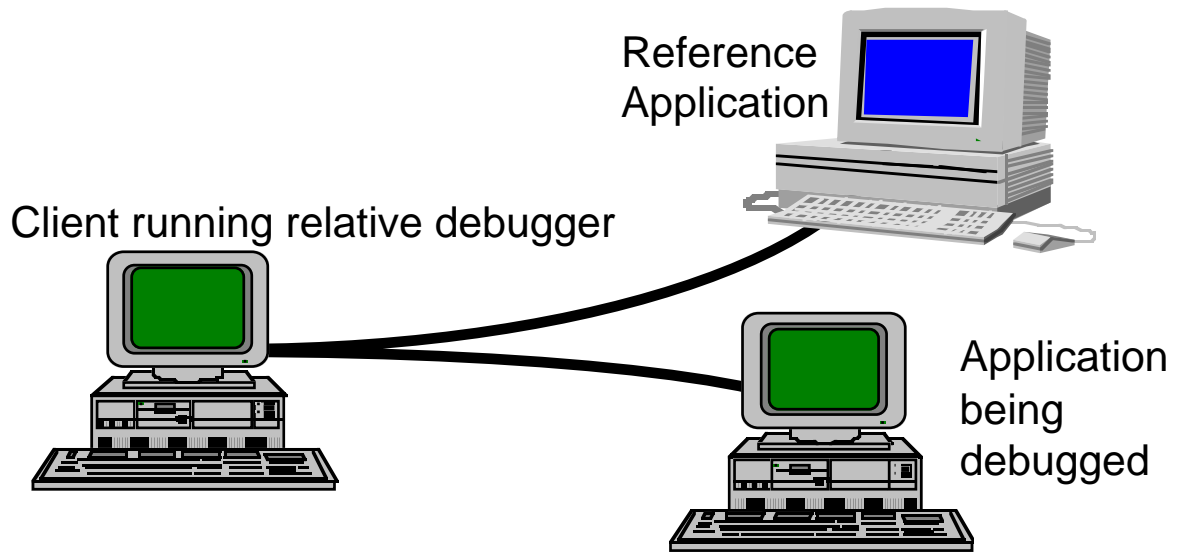


Figure 1 - A relative Debugger Environment

A relative debugger provides all the functionality of a traditional debugger, including commands for program control, state access and breakpoints. However, the heart of the relative debugger is a set of new commands, not available in conventional debuggers. These commands support the relative debugging methodology. We introduce them briefly here, and describe them in more detail in the next section when we discuss Guard.

Because the reference and development versions of the program are executed concurrently, a relative debugger must be capable of handling two programs at the same time. It is useful for the relative debugger to support the debugging of programs written in different programming languages and executing on different computers in a heterogeneous network, as shown in Figure 1. This makes it possible to use the relative debugger when porting programs from one language or computer to another. The implementation details of controlling multiple concurrent programs on a heterogeneous mix of machines is outside the scope of this paper, and is discussed in more detail in [3].

A relative debugger checks user-supplied assertions by comparing data structures in the reference and development versions. It performs necessary transformations of different internal data representations on different computers or in different languages. When performing comparisons, the debugger must take into account different data types, allowing for such issues as inexact equality in floating point numbers, and differences in dynamic pointer values. This aspect of the debugger will be illustrated in the next section.

Violations of assertions are reported to the user. A number of approaches are possible for reporting differences in data structures, ranging from text to advanced data visualization techniques. If there are only a few differences, then the numeric values of differences are printed out. If differences are numerous, then visualization techniques are required to present them in a meaningful way. Again, some example of the different techniques will be shown in the remainder of the paper.

### 3. The Guard Relative Debugger

The Guard relative debugger [1, 2, 3] provides standard debugging commands to start process execution, to set breakpoints, to examine process state, and so on. The main power and novelty of Guard, however, comes from its specialized commands that support relative debugging. These commands can broadly be broken into two groups, *process control* and *data comparison*.

#### 3.1. Program Control

Guard commands use symbolic names to refer to executing programs. Names are assigned when Guard launches a new program or when it attaches to one which is already running. Guard can handle an arbitrary number of programs concurrently, limited only by operating system restrictions.

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

```
invoke program_name command [machine_name] [user_name]
```

The `program_name` is a Guard variable which is then used to identify the running program. The `command` specifies the command line for the program. The optional `machine_name` and `user_name` make it possible to run the code on a remote system as another user. In this case, Guard creates and attaches to a remote process on another computer system. The ability to execute the reference and development versions of a program on different systems eliminates errors introduced by moving the reference code to the target system.

Processes created by `invoke` remain suspended until their execution is resumed by a `verify` command, which takes a list of program names as its argument. Execution continues until the processes terminate or encounter a breakpoint or an assertion.

#### 3.2. Data Comparison

Guard supports two different approaches to the comparison of program executions: a *procedural* approach and a *declarative* approach. We describe the procedural approach first.

The basic command for procedural relative debugging is `compare`, which has the following syntax:

```
compare <program_1>::<<variable_1> <program_2>::<<variable_2>
```

This command traverses two named data structures (`variable_1` and `variable_2`) from the two named programs and compares corresponding data elements, taking into account their data type. The user can specify a tolerance value for comparisons, in which case differences are reported only if two corresponding values differ by more than the tolerance. For array comparisons, `compare` interprets array indices according to the source language. This enables comparisons of multidimensional arrays between programs whose source languages use different layouts of rows and columns, such as Fortran and C. Currently, Guard only supports the comparison of elementary variables and arrays. Support for more complicated data structures such as records and dynamic data structures is planned in the future.

To use `compare`, the user must set up breakpoints in both processes, run the programs and issue a `compare` command when the programs reach the breakpoints. This procedure can be tedious and error prone, especially when comparing values within loops. To eliminate this laborious procedure, Guard provides declarative relative

debugging with assertions. Assertions simplify the debugging process by automating the setting of breakpoints and comparison of data structures.

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

```
assert [/eps type value] [/file filename] [/force]
    <program_1>::<variable_1>@<source_file_1>:line_1 =
    <program_2>::<variable_2>@<source_file_2>:line_2
```

The command contains two tuples, each with a program identifier, a variable name, a source file name and a line number. Guard plants breakpoints in the two programs at the specified line numbers, and stores the assertion information in an internal table. When breakpoints are encountered, the table is searched for matching assertions. The two data structures are then retrieved and compared. If there are no differences, then execution is resumed. If differences are detected, Guard returns to the command line, which enables the use of commands such as *print*, *compare*, etc. for examining process states.

The optional fields in the assertion make it possible to set tolerances for comparison of structures for each individual assertion (the */eps* parameter) and to specify a file for output of the differences (the */file* parameter). Data produced with the latter option can be used as input to a visualization program for a more detailed exploration of the differences. It is also possible to set global values for the tolerance and output file, which can be overridden using the */eps* and */file* parameters in a particular assertion. The */force* option causes the contents of the data structure in the reference code to be copied into the equivalent data structure of the code being debugged if the assertion fails. This feature makes it possible to determine whether the difference which is detected by a particular assertion is responsible for the failure of the overall program, without making any changes to the source of the programs.

### 3.3. Subarray Expressions and Index Permutations

By default, Guard compares all elements of array data structures. However, in many cases it is desirable to consider only a subsection of each array. For example, when comparing a sequential reference code with a parallel implementation, it is often the case that additional array rows and columns are added to the parallel code to facilitate the communication of boundary values.

Guard provides subarray expressions to allow the user to make an assertion on rectangular subarrays. The following example compares one array with a subarray of the same size from a larger data structure:

```
var1[1..61][1..61][1..23] = var2[6..66][6..66][1..23]
```

Guard also supports the permutation of array indices. This feature caters for cases where the data structures being compared are equivalent, but order their indices differently. The following syntax specifies that the second array uses an index ordering that is inverted relative to the first array:

```
var1[1..61][1..61][1..23] = var2[1..23][6..66][6..66]
/permute (2,1,0)
```

This feature has proved valuable when comparing versions of a code which are optimized for a vector architecture with those optimized for RISC ones. In these cases, the inner vector loops are often moved to outer loops to improve cache performance.

### 3.4. Visualizing Differences

Guard supports three approaches to the reporting of differences: text, bitmaps, and advanced visualization techniques through external visualisation utilities. Text output is the simplest; the actual values and differences are printed on standard output.

The second approach is more suited to array comparisons, where text output may be excessive. In this case, only two values are printed: the maximum difference between corresponding array elements, and the total cumulative difference between all elements. Most of the information is reported in a rectangular bitmap displayed on the screen. In this bitmap, white pixels denote values that are the same, and black pixels denote values that are different. This simple array visualization is particularly useful for detecting erroneous loop bounds and addressing expressions, because these types of error tend to generate regular patterns on the display. An example of such a visualisation is shown in Figure 2. In this case, a number of columns of the two arrays differ, and this is clearly seen as a black stripe on the right hand side of the array visualisation. Arrays with more than two indices can be folded onto two dimensions using a number of standard techniques.

The most powerful technique supported by Guard involves the use of commercial data exploration and visualization software such as IBM's Data Explorer (DX). A complete set of differences can be saved to a file using a parameter on the `assert` command. Values from the file can then be displayed using DX. This use of advanced visualization techniques is well suited to the display of differences in arrays with more than two dimensions. Animations can be used to convey the development of differences as the two programs execute. Figure 3 and 4 show a few examples of the visualisation of errors using DX. In both of these cases an error surface is plotted which shows the regions of three dimensional space in which the error exceeds a threshold. For example, in Figure 3 the difference between the temperature variable is shown in red when it exceeds 0.1% relative error. In figure 4, the red region indicates where error exceeds 10%, and the yellow region indicates where the error exceeds 5%. This image shows that the error only occurs in half of the space, which corresponds to a region which models the physics of pollution transport over land rather than water. The picture also shows that the error is transported vertically, implicating the physics code responsible for vertical advection. Displaying the error surface in this way provides a great deal of information about the potential source of the error, including its possible location. For example, the position of the error may exclude some routines of the code which do not manipulate that region of space.

### **3.5. Underlying Guard Technology**

Space does not permit a detailed description of Guard implementation. However, we provide a few relevant details. Guard is implemented above a platform, called Dynascope, which provides an interface for building debuggers [9, 10]. The interface contains operations for process control, state access and breakpoint handling.

Dynascope's (and hence Guard's) machine independent debugging interface is provided through debugging servers [10]. A debugging server is associated with each executing program. This server executes as a separate process or as part of the program, and is responsible for receiving and executing debugging requests from Guard. The communication between servers and Guard is performed by TCP/IP.

Because Guard is implemented in terms of Dynascope mechanisms, Guard can be ported with little modification to any computing platform supported by Dynascope. (Currently, debugging servers have been implemented for Sun, Silicon Graphics, IBM RS6000, and Next computers). In addition, the support for remote debugging allows Guard to compare programs executing on different machines. For example, we have successfully run Guard across the Internet, with Guard executing on a Silicon Graphics computer in Australia, and the two programs that were being compared running on a Sun computer in Europe and a Next computer in the U.S.A.

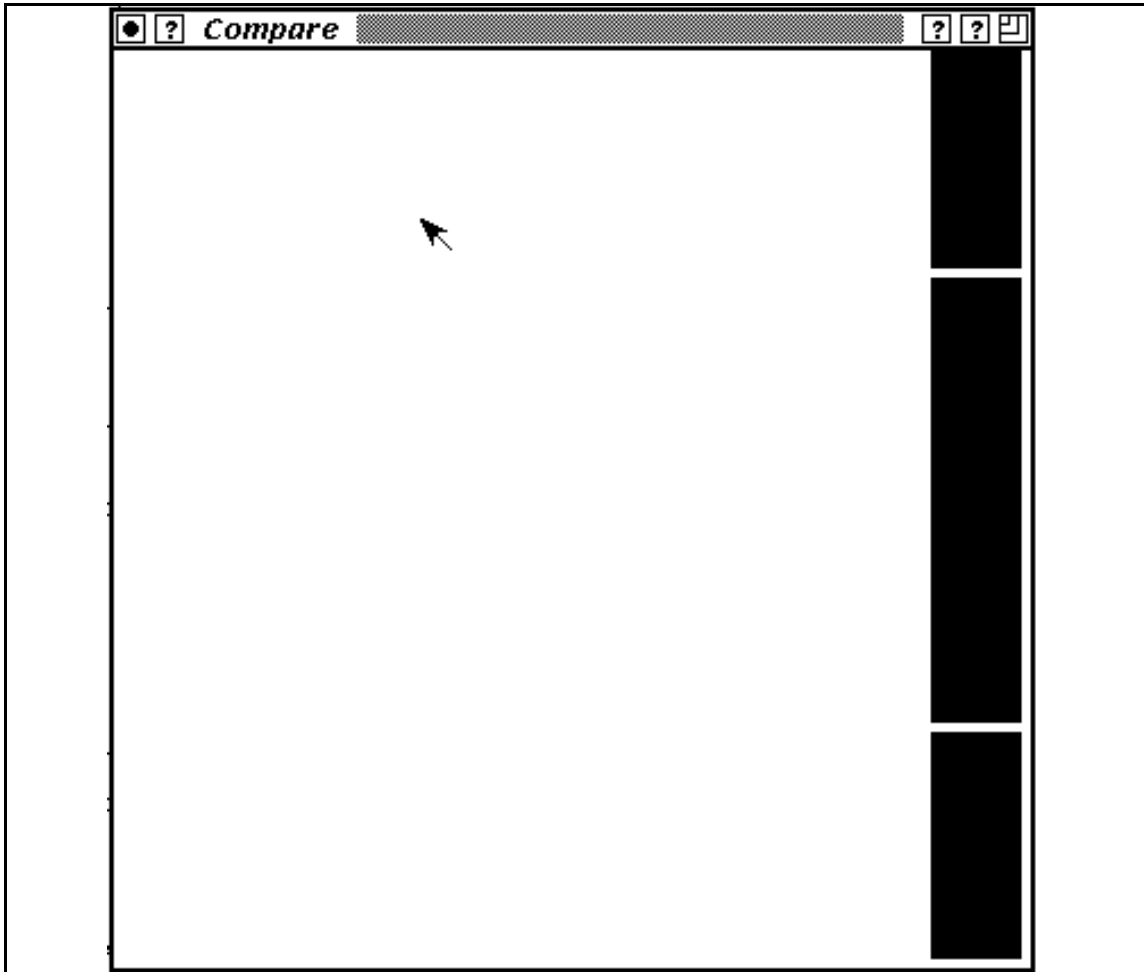


Figure 2 - Visualisation of boundary errors in two dimensional arrays, produced by Guard's inbuilt array visualisation system.

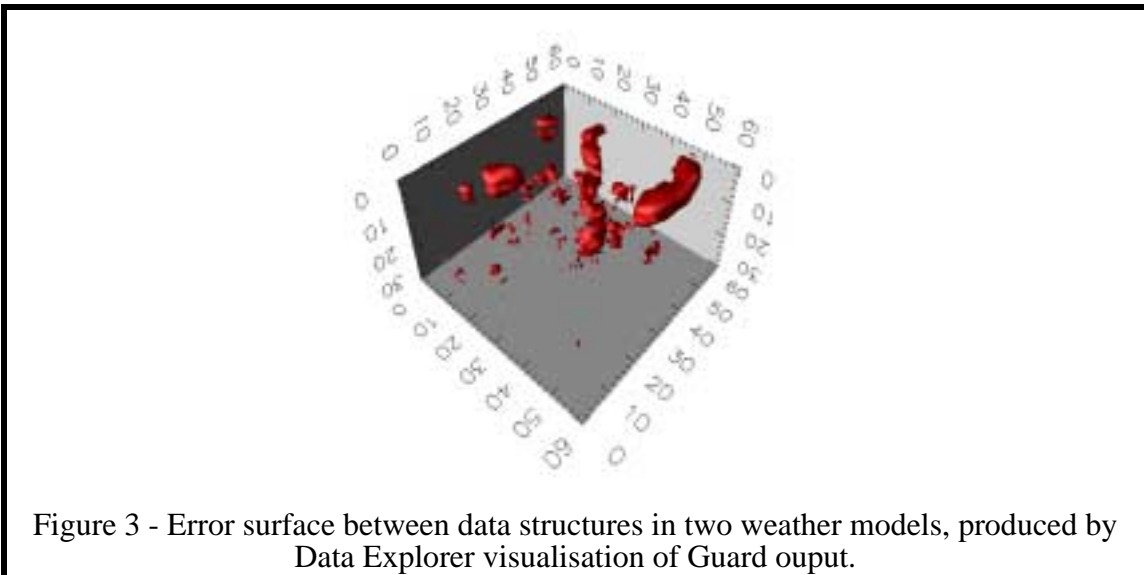


Figure 3 - Error surface between data structures in two weather models, produced by Data Explorer visualisation of Guard output.

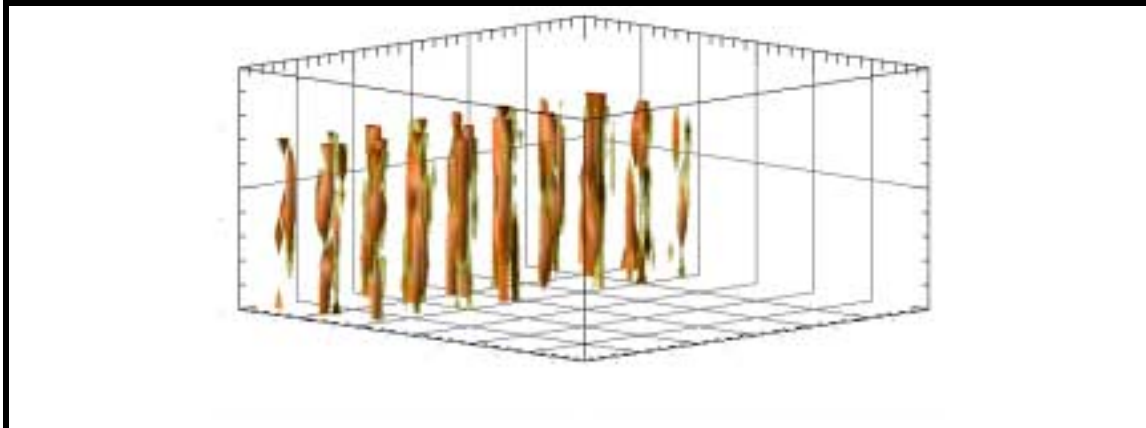


Figure 4 - Error surface showing difference in ozone concentration between two photochemical pollution models, produced by Data Explorer visualisation of Guard output.

## 4. Test Cases: Atmospheric Modelling Applications

In this section we describe the experiments performed using Guard to trace discrepancies in MM5, a large meso-scale weather model. First, we describe an experiment in which Guard was used to determine that two sequential versions of MM5, thought to implement different dynamics, in fact provided identical results. Second, we report an experiment in which Guard was used to investigate and account for discrepancies between a correct sequential version of MM5 and a single process parallel version called MPMM. This task was particularly challenging because of the different data and code structures. More details of MM5 and MPMM are provided in the side bars.

### 4.1. Comparing Sequential Codes

In this experiment we used Guard to compare two versions of the sequential MM5 code. The first version was a Unix code produced at Argonne from the original NCAR Cray code; the second was a code produced at NCAR that incorporated refined versions of a number of dynamics and physics routines. The two codes were supposed to be functionally equivalent, but we did not expect bit-for-bit correspondence. Our task was to quantify any discrepancies between the two models, and to isolate the source of these discrepancies.

We proceeded as follows. Guard was configured to compare a number of the principal MM5 data arrays. Initially, we ran the models for 30 time steps with a small tolerance value. There were no errors reported. Further, the errors were all zero, indicating that the codes were producing exactly the same results. To our surprise, Guard had indicated that the two models were providing bit-for-bit identical results.

Although simple, this experiment provides some encouraging information regarding Guard's capabilities. First, it confirmed Guard's ability to deal with large, complex codes. MM5 is over 30,000 lines in length, and the data structures that it operates on are also large. Second, the ease with which we could perform the comparisons, and look for both tolerance-based differences and nonidentical values, was impressive. Third, the computational overhead of using Guard proved to be (as expected) minimal relative to the execution time of the numerical model under examination. Finally, the experiment was performed by one of the authors who had no prior knowledge of the structure or operation of MM5. In all, it required about 1 hour of time from the initial examination of the code to the specification of the assertions until the results were delivered.

## 4.2. Comparing Parallel and Sequential Codes

In the second experiment reported in this paper, we used Guard to compare the sequential MM5 and a single-process implementation of the parallel MPMM. As in the preceding case, we started with the assumption that the code was incorrect. The challenge was to validate this assumption and then to determine the source(s) of any discrepancies.

The preliminary set of assertions, shown in Figure 5, was applied to major data structures showed a number of differences. Commands 1 and 2 start the two different versions of the model. Command 3 sets the tolerance to ignore minor numeric differences and allows the user to focus on significant errors only. Command 4 specifies an assertion for comparing the data structure `ta` in the new and old models. Note that sub arrays are used to extract the correct regions of the arrays for comparison, and that index permutation is required because the two arrays are ordered differently. The differences were initially small and grew slowly. Visualization showed significant spatial structure, suggesting that they were not due solely to floating point rounding. Incorporating further assertions allowed us to identify the planetary boundary layer scheme (subroutine HIRPBL) as the source of at least one error. Assertions applied to the input arguments to this routine showed only minor differences, concentrated around the boundary of the grid; assertions applied to the output arguments showed larger and more widespread differences (Figure 6).

```
1.  invoke new mpmm
2.  invoke old mm5.exe
3.  eps relative float .001
4.  assert old::ta[1..60][1..60][1..23]@SOLVE3.f:157 =
      new::ta[1..23][6..65][6..65]@compute_solve3.f:908
      /permute (1,2,0)
```

Figure 5 - Assertions used to trace errors in two models.

Having localized the error to the HIRPBL routine, we switched to a manual code inspection. We soon identified the source of the discrepancy. Briefly, the problem turned out to be a difference between the way the serial and parallel models determined the number of minor loop iterations required for a vertical column of air at each grid point. For a given point, this number of iterations can vary from a minimum of 2 to a maximum of 13 (for this problem) and the difference is subject to terrain and conditions of the atmosphere.

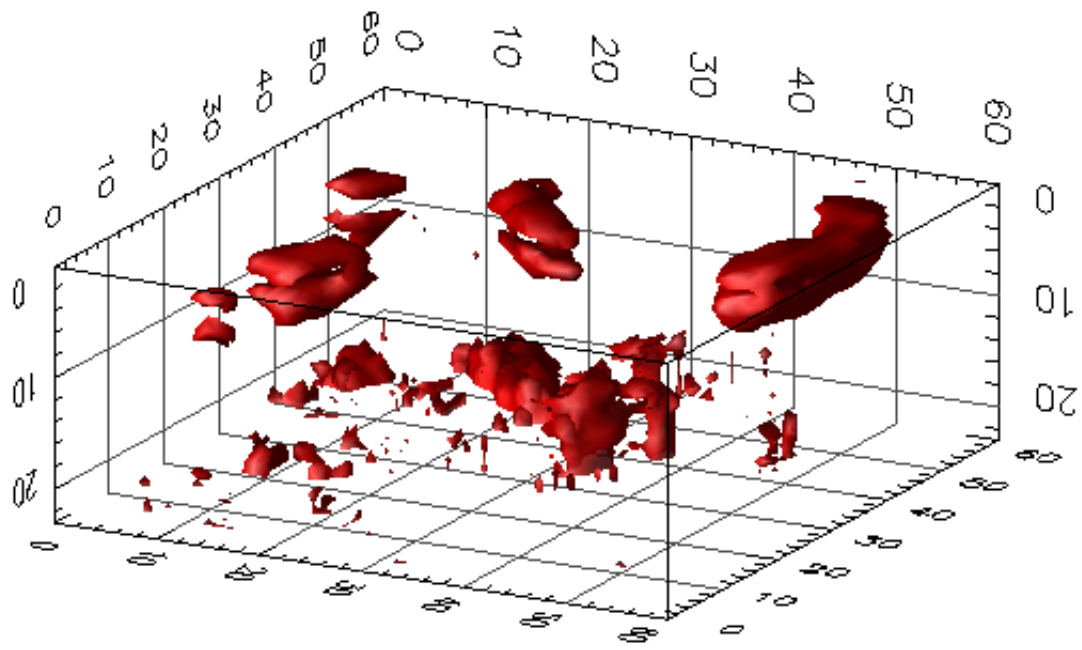


Figure 6 - Temperature error surface showing error  $> 0.1\%$

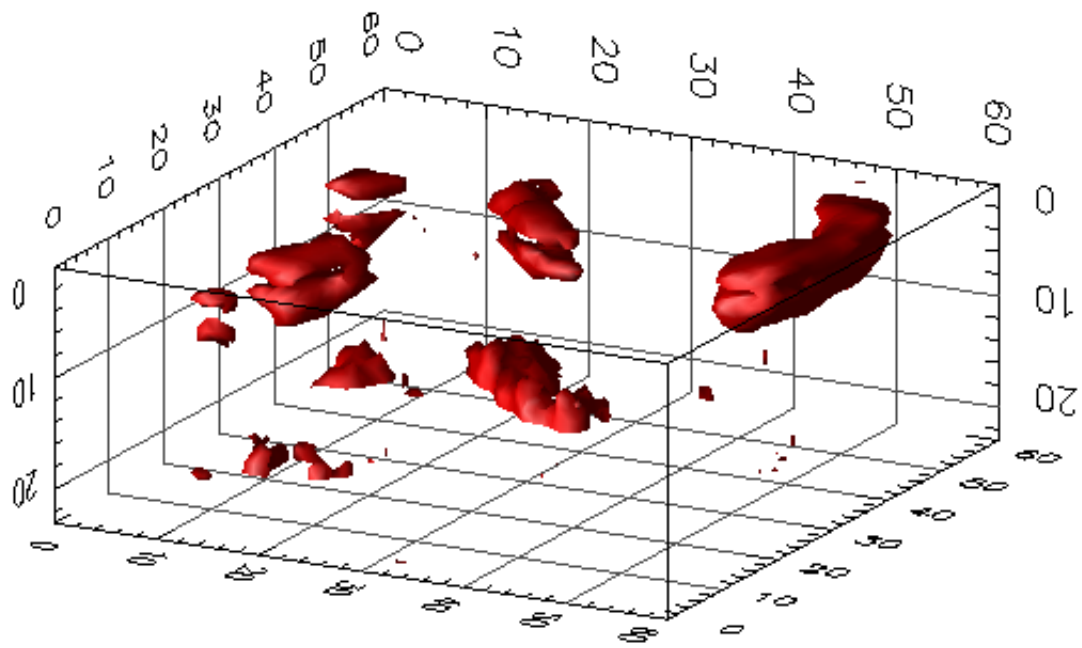


Figure 7 - Temperature error surface when 13 iterations used by both codes

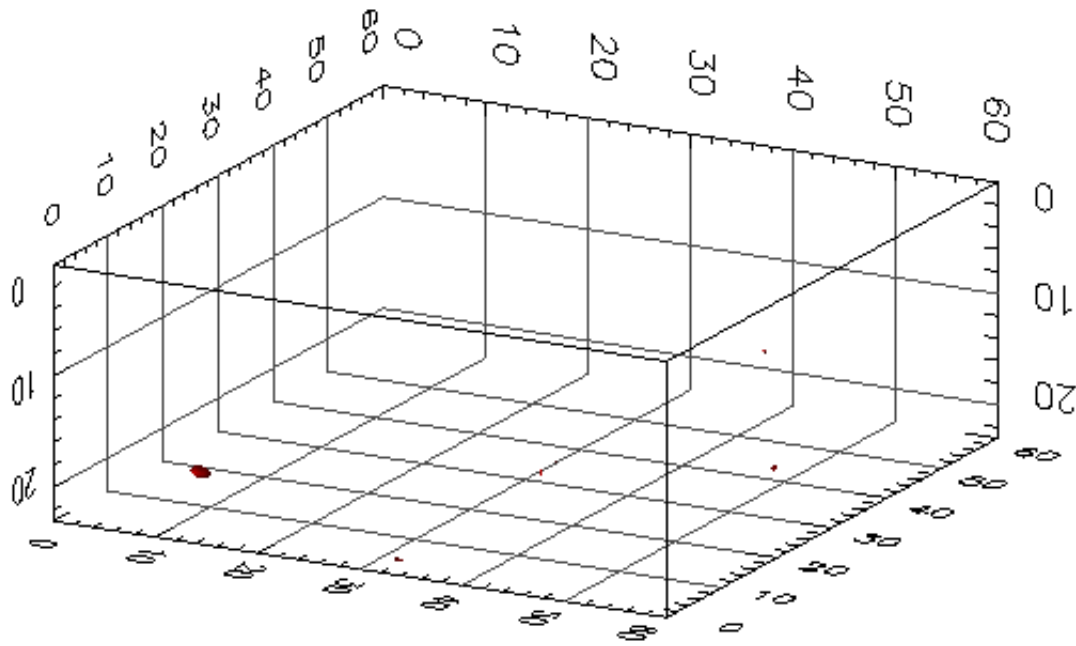


Figure 8 - Temperature error surface after term added into long wave radiation term

In order to maximise the performance on a vector supercomputer, the sequential code first determined the maximum number of minor iterations over an entire north-south strip, and then performed that number of iterations for each column in that strip. Applying this same strategy in the parallel code would introduce an unnecessary interprocessor communication (to determine the maximum number of iterations) and work. Hence, the parallel code performed just the required number of iterations for each column.

Suspecting that this difference in behavior was the source of our problems, we modified both the sequential and parallel codes so that both performed 13 iterations on all columns. Comparison of the modified codes using Guard showed that the discrepancy over the interior of the grid had disappeared (Figure 7). This result led us to conclude that the observed discrepancy did not reflect an error in the parallel code, but rather was the consequence of a known and allowable difference in the model codes.

The modifications to HIRPBL did not remove all differences between the two models, as seen in Figure 7. Further analysis with Guard showed that there was a term missing from an equation in the long wave radiation code in MPMM (in routine LWRAD), and thus the two models were computing different amounts during this phase. When the term was added to MPMM almost all of the remaining differences were removed, as shown in Figure 8.

Figure 9 shows a front view of the error surface after 45 time steps. Two separate regions have been highlighted. The picture clearly shows an error region at the bottom of the atmosphere, and another one at the top. As discussed, these divergences came from different regions of the model.

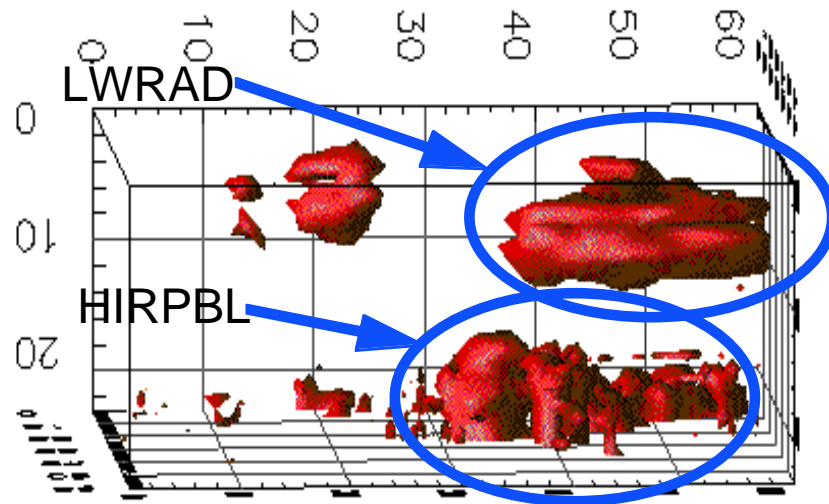


Figure 9 - Front view of error surfaces identifying LWRAD and HIRPBL contributions to divergence

In summary, in this experiment Guard allowed us to account for a number of discrepancies between the sequential and parallel codes.

## 5. Conclusions

Relative debugging is a new debugging methodology for applications which undergo evolutionary changes; Guard is a debugger that supports this methodology. The key idea is to provide support for the automatic comparison of program state with the state of a reference program that is known to be correct. Relative debugging appears to be particularly useful in a scientific computing context, because of the complexity of scientific models and the frequent need to adapt existing models to incorporate new physics, algorithms, or computational techniques.

We have applied Guard to a large scientific code, the MM5 mesoscale atmospheric model, and obtained very satisfactory results. We were able to use Guard both to verify that two versions of the sequential MM5 were functionally equivalent, and to isolate discrepancies between sequential and parallel versions of MM5.

We are currently working on a number of extensions to Guard. One important direction which will significantly increase its utility is support for the debugging of parallel programs. Another direction, motivated by the MM5 experiments described in this paper, is to provide support for more flexible comparison of codes with different control structures. A commercial version of Guard is also under development.

## References

- [1] Abramson and R. Sasic, "A Debugging and Testing Tool for Supporting Software Evolution", to appear, *Journal of Automated Software Engineering*.
- [2] Abramson and R. Sasic, "A Debugging Tool for Software Evolution", *CASE-95, 7th International Workshop on Computer-Aided Software Engineering*, Toronto, Ontario, Canada, July 1995, pp 206 - 214.
- [3] Abramson, D.A., Sasic, R. and Watson, G. "Implementation Techniques for a Parallel Relative Debugger ", to appear, *International Conference on Parallel*

Architectures and Compilation Techniques - PACT '96, October 20-23, 1996, Boston, Massachusetts, USA.

- [4] Galbreath, W. Gropp, and D. Levine. "Applications-driven parallel I/O.", In *Proceedings Supercomputing-93*, Portland, Oregon, pages 462-471. IEEE, 1993.
- [5] May J. and Berman, F. Panorama: A Portable, Extensible Debugger, ACM/ONR Workshop on Parallel and Distributed Debugging, Sigplan Notices, Vol 28, No 12, 1993, pp 96-106
- [6] Maybee, P. NeD: The Network extensible debugger, Proceedings of the Summer 1992 USENIX Technical Conference, San Antonio, 1992
- [7] Olsson, R. Crawford, R. Ho, W. A Dataflow Approach to Event-based Debugging, *Software Practice and Experience*, Feb 1991, Vol 21, No 2, 209-229
- [8] Ramsey , N. and Hanson, D. A Retargetable Debugger, Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation, 1992, ACM Press, pp 22-31
- [9] Sosic, R. A Procedural Interface for Program Directing, *Software Practice and Experience*, Vol 25, No 7, July, pp 767-787, 1995
- [10] Sosic, R. Design and Implementation of Dynascope, a Directing Platform for Compiled Programs, *Computing Systems*, Vol 8, No 2, Spring, pp 107-134, 1995

## **Acknowledgments**

The development of MPMM was supported in part by the USAF and the EPA. We are grateful to Bill Kuo, Jimmy Dudhia, and Georg Grell for making MM5 available to us. Some of the experimental studies were performed on the IBM SP2 at Argonne National Laboratory and the IBM SP2 at the Queensland Parallel Supercomputing Facility. We also express our gratitude to Andrew Lewis for assisting with data visualization. Development of Guard was supported in part by the Australian Research Council and IBM Australia.

The paper reports a collaborative project between Griffith University and Argonne National Laboratory. Guard was developed at Griffith University during 1994. The experiments reported in the paper were mostly performed at Argonne during March 1995. The work has allowed us to demonstrate the utility of Guard on large scientific codes, and has motivated numerous enhancements to Guard.

## Side Bar 1: MM5

The Penn State/NCAR Mesoscale Model is a public domain limited-area primitive equation model designed to simulate meso-alpha scale (200-2000km) and meso-beta scale (20-200km) atmospheric circulation systems [1, 2]. Developed and maintained by the Pennsylvania State University and the National Center for Atmospheric Research, the model is used for real-time regional weather forecasting, air-quality studies, regional climate prediction, and other areas of atmospheric research.

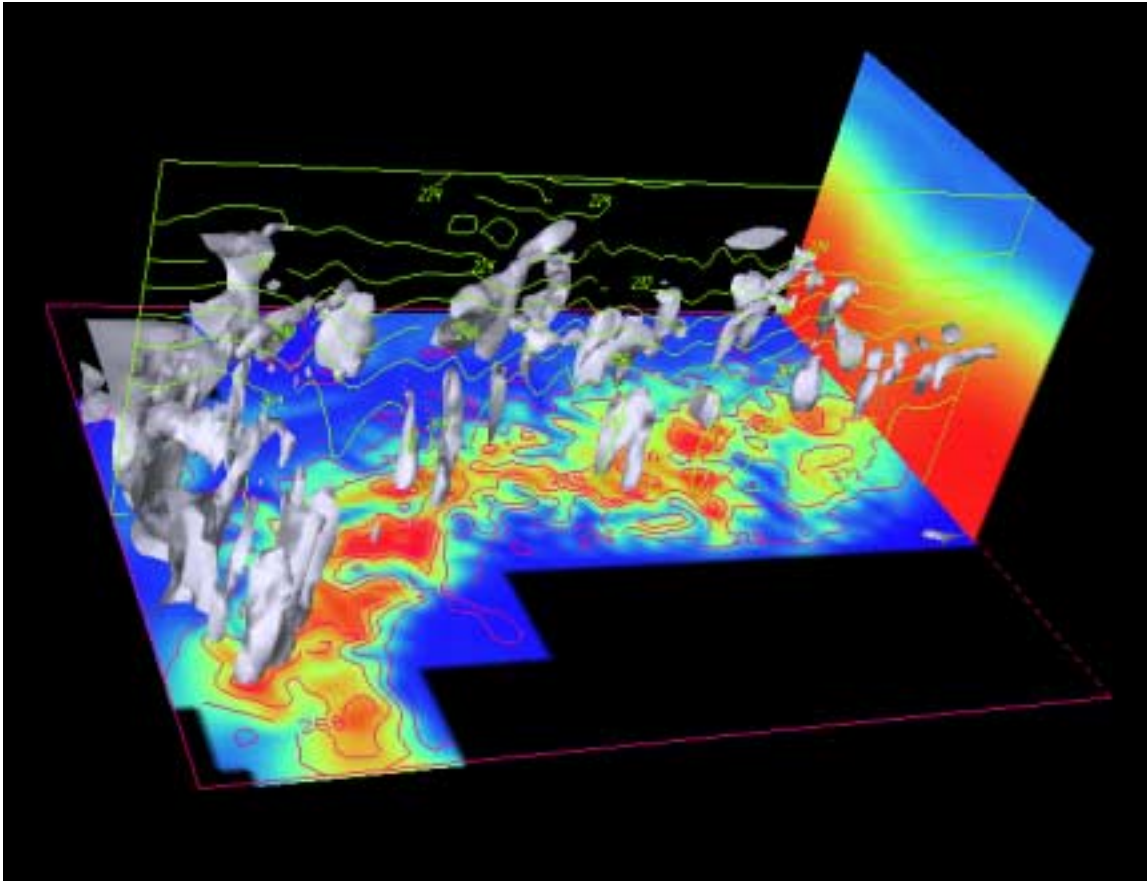
An MM5 domain is a horizontally-uniform rectangular grid representing three-dimensional regions of the atmosphere. The vertical coordinate system is terrain-following, with layers distributed more closely to the surface. A large number of physical parameterizations, collectively called "physics," provide forcing terms for solar radiation, cloud formation, surface and boundary layer effects, etc. Input to the model comes in the form of initial conditions - derived from observations, other simulations, or both - and lateral boundary conditions that are input periodically. The model also can also input observations that are scattered over time to further improve the quality of the simulated output. MM5 incorporates a sophisticated nesting mechanism for mesh refinement over complex terrain or developing systems, providing the ability to resolve to a horizontal resolution as small as several hundreds of meters. Nests may be overlapped and even moved, if necessary, over the course of a simulation.

MM5 is written almost entirely in standard Fortran 77. (However, pointers are used to keep track of different copies of model data that correspond to different nesting levels). Loops and data structures in the MM5 code are structured for good performance on traditional vector supercomputers such as those built by Cray Research Inc. Thus, although logically the code can be thought of as invoking physics routines on a set of independent vertical columns, for maximum vectorization the code is structured so that the longitudinal loop is innermost.

### References

- [1] Anthes, "1986 summary of workshop on the NCAR Community Climate/Forecast models.", *Bull. Amer. Meteor. Soc.*, 67:194-198, 1986.
- [2] Georg A. Grell, Jimmy Dudhia, and David R. Stauffer., "A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5).", Technical Report NCAR/TN-398+STR, National Center for Atmospheric Research, Boulder, Colorado, June 1994.

## Side Bar 2: MPMM



*MPMM simulation of an precipitation event, January, 1991. Output is from a 5km irregularly shaped nest incorporating high-resolution terrain data for a region over the Alps. The model was run on 12 processors of an IBM SP2. The visualization is using the VIS-5D package from the University of Wisconsin in Madison.*

MPMM is a version of MM5 designed to exploit scalable parallel computers [1, 2]. It is intended to be functionally equivalent to MM5 (that is, it should yield the same results), but is structured quite differently in order to execute efficiently on parallel computers. It has been developed by researchers in the Division of Mathematics and Computer Science at Argonne National Laboratories in Chicago, and whilst it is based on the original MM5 code, it has a wide number of differences.

Two significant concerns that arise when moving a vector code such as MM5 to a scalable parallel computer are single processor performance and load balance. MPMM is structured to address both these issues. The MM5 code is restructured to obtain a *column-callable* form, which permits the model to be called once for each vertical model column. A portable runtime system library (RSL) is then used to invoke the column-callable model once for each column. This approach facilitates load balancing, as it is easy to move individual columns between processors in response to changing load distribution. It also results in better cache utilization. The resulting code performs well on scalable parallel computers, achieving for example 2 GFlops on a 64-processor IBM SP2 with Power 2 processors, on a 61 by 61 by 23 domain.

## References

- [1] I. Foster and J. Michalakes. "MPMM: A Massively Parallel Mesoscale Model.", In Geered-R Hoffmann and Tuomo Kauranne, editors, *Parallel Supercomputing*

*in Atmospheric Science*, pages 354-363. World Scientific, River Edge, NJ 07661, 1993.

- [2] J. Michalakes, T. Canfield, R. Nanjundiah, S. Hammond, and G. Grell., "Parallel Implementation, Validation, and Performance of MM5", In *Parallel Supercomputing in Atmospheric Science*. World Scientific, River Edge, NJ 07661, 1994.