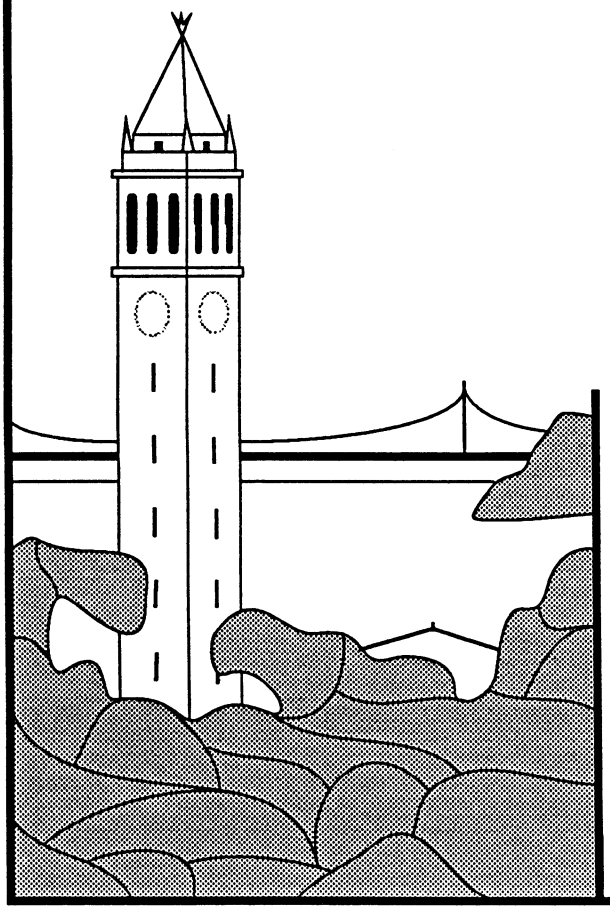
Mantis:
A Debugger for the Split-C Language

Steven S. Lumetta



Report No. UCB/CSD-95-865

February 1995

Computer Science Division (EECS)
University of California, Berkeley
Berkeley, CA 94720

University of California
Berkeley

Mantis:
A Debugger for the Split-C Language

a thesis submitted in partial satisfaction
of the requirements for the degree of
Master of Science in Computer Science

by

Steven S. Lumetta

February 7, 1995

Acknowledgements

First, I wish to thank my advisor, Professor David Culler, for his continued support and encouragement throughout the implementation of Mantis and the writing of this document. Without his encouragement, the debugger might still be just a dream.

My thanks also go to my other committee member, Professor Lawrence Rowe, whose teaching and comments greatly aided my understanding of user interface design and helped to shape the debugger.

Seth Goldstein and Andrea Dusseau, my officemates and friends, gave helpful suggestions on all aspects of the project, from implementation to writing to motivation, and for these I am grateful. Jeffrey Jones and Michael Mitzenmacher helped with the preparation of the thesis, giving freely of their time to provide editorial comments. I am also grateful to the members of the Spring 1994 CS267 class and its instructor, Professor James Demmel, as well as Arvind Krishnamurthy, without whose endurance Mantis might never have passed beyond the alpha version.

I wish next to thank the National Science Foundation and Lawrence Livermore National Laboratory, who supported this work through a Graduate Research Fellowship, Infrastructure Grant number CDA-8722788, and Grant number LLL-B28 3537.

Finally, and most importantly, I thank my wife, Jennie, whose love and patience with me have known no bounds during this very stressful period of my life. I am forever grateful for her companionship and support.

Abstract

Programming parallel machines need not be hard. A large portion of the computer science community is working to develop tools to ease parallel programming. Split-C, a language developed as part of the Castle effort at Berkeley, extends C to control parallel machines but retains the straightforward translation from source code to executable code necessary for high-performance programming. Mantis, the Split-C debugger, supports the bulk synchronous and individual node viewpoints that together dominate the design of Split-C programs. Mantis combines the flexibility and extensibility of a Tcl/Tk graphical user interface with the stability and functionality of the `gdb` *debugger, providing a simple but highly effective tool for parallel debugging. The user can control execution for all nodes as a group or for each node individually, and can check state and invariants through a variety of methods. The graphical interface is simple enough for new users to understand with minimal effort yet powerful enough to allow experienced users to work effectively. The first version of Mantis runs on the CM-5 and made its debut in the parallel programming course at Berkeley during the Spring 1994 semester.*

This document describes Mantis and our experiences in designing and implementing a parallel debugger. Using a straightforward example, we illustrate the process of using Mantis to find both simple and more subtle bugs. After summarizing the features of Mantis, we discuss a range of issues in parallel debugging and our approach to those issues. We follow with our experiences in working with large software systems, in particular the `gdb` *debugger. We continue with a discussion of user interface design, and conclude with commentary on related work and directions for future efforts.*

Contents

Abstract i

Table of Contents iii

List of Figures vii

1 Background 1

1.1 Introduction 2

1.2 Summary 2

2 Overview of Mantis 5

2.1 Goals and Programming Model 5

2.2 Illustration of Use 6

2.2.1 Finding a simple bug 7

2.2.2 Locating a more subtle bug 11

2.3 Summary of Features 14

2.3.1 Interface highlights 14

2.3.2 Window management 17

2.3.3 Parallel process control 18

2.3.4 Source browsing 19
2.3.5 Individual nodes and state 20
2.3.6 Data display and entry 20

3 Issues in Parallel Debugging **23**

3.1 Problem Localization 23
3.2 Breakpoints 24
3.3 Expression Evaluation Context 27
3.4 Race Conditions 30
3.5 Active Messages 31
3.6 Procedure Calls 32
3.7 Linear Time Operations 32
3.8 Asynchronous Notification of Status 33
3.9 Summary 33

4 Experience with Large Software Systems **35**

4.1 Extensions Required for Mantis 36
4.2 Effect of Debugger Model on Implementation 37
4.2.1 Ideal set of changes 37
4.2.2 Changes required for the CM-5 version 39
4.2.3 Complications of the single process debugger model 41
4.2.4 Asynchronous notification revisited 42
4.3 Experience and Tradeoffs 43

5 User Interface Design **45**

5.1 Simple, but Efficient 46
5.2 Instantaneous Reaction 46
5.3 Intelligent Prediction 47
5.4 Caching Data 49
5.5 Support for Known Interfaces 49
5.6 Efficient Information Display 50
5.7 Common Design Pattern 51
5.8 Pleasing to the Eye 52
5.9 Building Interfaces with Tcl/Tk 53

6 Related Work **57**

6.1 Other Parallel Debuggers 57
6.1.1 Panorama 57
6.1.2 Node Prism 59
6.1.3 TotalView 66
6.2 Different Approaches to Debugging 67
6.2.1 Tracing communication 67
6.2.2 Animation 68
6.2.3 Dynamic instrumentation 69

7 Future Work **71**

7.1 Extensions to the Interface 71
7.2 Library Support 72
7.3 Mantis as a Subprogram 73

7.4 Porting to Other Platforms 73

A Code for Fish and Gravity **77**

Bibliography **83**

List of Figures

2.1 Bulk synchronous block diagram. The main loop of the sharks and fish program cycles through three blocks. 6

2.2 Main window. Symbols for the `fish` program have just been loaded. 7

2.3 Status window. Many nodes are still running (green/dark) and many others have errors (yellow/light). 8

2.4 Node window. Node number 46 encountered a bus error in `all_compute_force` at the line highlighted in black. 9

2.5 Evaluation window. Examining the expression "local_fish" reveals the cause of the problem. 10

2.6 Output window. The partially debugged program hangs before completion. . 10

2.7 Node window. Node 0 hung in a barrier. The programmer moved up the stack to `splitc_main`, and the `barrier` call is highlighted in black. 12

2.8 Local variables window. The variables shown correspond to the `splitc_main` stack frame chosen in the node window. 13

2.9 Display window. The expression "delta_t" is evaluated on node 0 each time the processor stops or the programmer changes the frame. 13

2.10 Another display window. The expression "delta_t" is evaluated on node 1. . 14

2.11 Global window. The programmer set a breakpoint at line 99 in `splitc_main`. 15

2.12 Window hierarchy. The output window appears automatically when the user's program has output. 17

3.1 Stack trace overview. The debugger displays only differences in stack frames, maintaining the content but reducing the volume of information given to the user. 24

3.2 Mismatched task completion. The programmer's conception of task completion appears on the left (a), and actual task completion appears on the right (b). 25

3.3 Bulk synchronous block diagram. The italicized text explains how the bug violated the block structure. 26

3.4 Data types in the Split-C address space. The dashed lines indicate that the automatic variable is only accessible from within a certain procedure 28

4.1 Ideal implementation structure. The node debuggers use a standard sequential interface to the child processes. The global debugger provides machine-wide functionality. 37

4.2 CM-5 implementation structure. A single process combines the global and node debuggers, interacting with the child processes via the Time Sharing Daemon 39

5.1 Evaluation window. A pull-down menu displays the cache of previously evaluated expressions. 48

5.2 Main window. Control buttons found to the right of entry boxes are used for menus, file selection dialogs, and actions associated with the entry text. . . . 51

5.3 Node window. The source file display widget directs the user's attention to a problem. 54

6.1 Main window of Node Prism. The source region automatically displays the start of `main`. 60

6.2 File selection window of Node Prism. Files with inlined code such as `bulk.h` appear once for each instance of inlining. 63

6.3 Stack trace comparison window of Node Prism. The second bug from Chapter 2 causes the first branching. 65

7.1 High-level language built using Split-C and Mantis. The user need only interact with the high-level compiler and debugger. 74

Chapter 1

Background

Programming parallel machines need not be hard. We can solve the difficulties of parallel programming just as we have overcome other "impossible problems" in computer science, by developing simple and easy-to-use tools.

A large portion of the computer science community is working towards solutions for parallel programming. At Berkeley, the Castle project [1] is building an integrated parallel programming environment from the ground up, from low-level interprocessor message libraries that deliver messages at speeds close to hardware latencies to high-level parallel languages that support the data abstractions necessary for parallel programs. Split-C is one of the lower levels of the Castle system, sitting just atop the Active Message communication layer [21]. Split-C extends C to control parallel machines but retains the straightforward translation from source code to executable code necessary for high-performance programming.

We first implemented Split-C on the Thinking Machines Corporation CM-5, and before the implementation was stable, we began to write programs. We soon realized that the tools available at the time were not practical for debugging Split-C programs. The tools gave no support for the extensions to C, and they provided poor user interfaces. Despite the fact that Split-C compiled much more quickly than other parallel languages, the lack of debugging facilities began to take its toll. In some cases, we were forced to spend days searching for bugs using `printf` and recompilation to perform a binary search on the code. We soon agreed that Split-C was stable enough to begin development of the environment and tools to support higher levels of the Castle, and we decided to build a debugger for the language. Our efforts resulted in Mantis, the Split-C Debugger.

1.1 Introduction

The first version of Mantis runs on the CM-5 and made its debut at Berkeley during the Spring 1994 semester. The students in the parallel computation course used Mantis to debug their final projects. Mantis combines the flexibility and extensibility of a Tcl/Tk [16] graphical user interface with the stability and functionality of the `gdb` debugger, providing a simple but highly effective tool for parallel debugging.

We chose to work with `gdb` to avoid the time required to code the generic sections of the debugging platform. Despite the difficulty involved in learning the internals of a large software system like `gdb`, we believe that we developed a more stable and more portable parallel debugger in less time than we could have managed by writing all of the code ourselves.

The Mantis graphical user interface provides a hierarchy of functionality. The main window controls file selection and window management. A status window gives an efficient, graphical representation of the status of all processors, allowing the user to locate problems rapidly and to investigate with a mouse click. The debugger allows the user to control all processors simultaneously or to focus on a single processor, supporting both the bulk synchronous and locally sequential views typical of Split-C programs. Data display windows allow the user to examine and change state.

1.2 Summary

This document describes Mantis and our experiences in the design and implementation of a parallel debugger.

We begin Chapter 2 with a description of the programming model used in Split-C. With this model in mind, we transform general debugger goals into specific goals for Mantis. We then explore the features and capabilities of the debugger by finding two bugs in an example Split-C program. The chapter concludes with a summary of debugger and interface functionality.

In Chapter 3, we touch on a number of issues in parallel debugging, indicating how choices of language, system software, and hardware affect debugging concerns. We focus on the requirements for Mantis on the CM-5, and give the details of our solution to each problem.

Chapter 4 covers our experiences in working with large software systems, in particular the Free Software Foundation's `gdb` debugger. We begin with a general description of the `gdb` system, then proceed with a list of changes required to create a generic parallel debugger.

We propose an ideal model of the debugger and show how that model minimizes the effort required to build a parallel debugger with `gdb`. We then discuss how restrictions imposed by the CM-5 operating system alter the model and increase the required effort. After investigating the ramifications of the change in model, we conclude with a summary of the tradeoffs involved in using a large, pre-existing software package like `gdb` as a base instead of writing all of the code necessary for a parallel debugger.

We explore user interface design in Chapter 5, applying general notions of what makes a good interface to specific problems in the design of Mantis. We conclude with a discussion of the Tcl/Tk package [15][16][17] used to create the Mantis user interface.

Chapter 6 offers a discussion of related work. We begin with commentary on other parallel debuggers, including a highly portable research debugger and two commercial parallel debuggers. We find that debugger functionality plays an important role in overall usefulness, but that a poor user interface can effectively cripple the program. We also review other methods of debugging in this chapter, including communication tracing and deterministic replay schemes and brief overviews of animation and dynamic instrumentation.

Chapter 7 outlines directions for future work, including extensions to the interface and the addition of debugging support to the Split-C library. We discuss the possibility of using Mantis as a building block for high-level language debuggers, and end with our plans to port Mantis to other platforms.

Chapter 2

Overview of Mantis

This chapter gives an overview of Mantis, introducing the goals, interface, and features of the debugger. Section 2.1 discusses the goals of the debugger and the general Split-C programming model. Section 2.2 walks the reader through an example of using Mantis to find bugs in a simple piece of code. Section 2.3 summarizes the features of Mantis.

2.1 Goals and Programming Model

Before exploring Mantis, ask yourself this question: what should a debugger do? In response, we offer the following: the debugging environment must support the programmer's conception of the program by allowing the same set of abstractions and the same viewpoint used in the compilation environment. It must also provide efficient means of performing common tasks such as execution control (e.g., breakpoints) and verification of invariants. What do these concepts mean in terms of Split-C?

In our experience, most Split-C programs can be decomposed into two layers. The top layer consists of a set of bulk synchronous blocks. The term "bulk synchronous" implies that the computing nodes enter each block more or less simultaneously and synchronize at the end of each block before entering the next. The bottom layer occurs within blocks. At this layer, the programmer thinks of nodes individually—each operates on a different set of data, or possibly executes different code, but the model is locally sequential.

For the debugging example in this chapter, we draw on code to simulate the world of WaTor, introduced by A. K. Dewdney in 1984 [5] and documented further by Fox et. al. [7], which has been a valuable tool for teaching parallel programming at Berkeley (in the CS267 course). In the original WaTor, sharks and fish share a world of water and interact

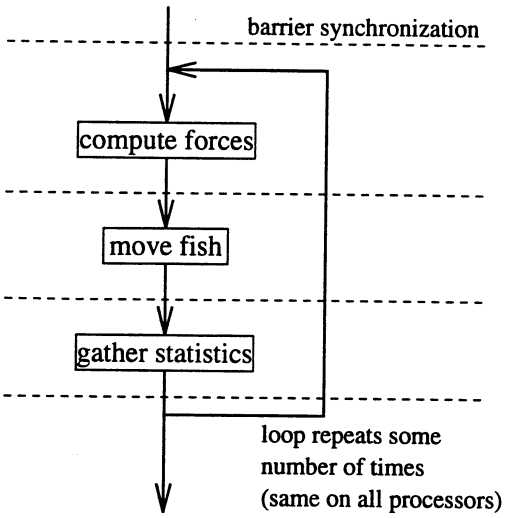


Figure 2.1: Bulk synchronous block diagram. The main loop of the sharks and fish program cycles through three blocks.

through a small set of rules. We use a version of WaTor in which fish alone populate an infinite plane and attract other fish according to an inverse-square law. This version of the problem introduces programmers to basic issues in data distribution and access as well as typical methods used in solving gravitational and electromagnetic problems.

The commented code in Appendix A presents a solution illustrating the Split-C programming model discussed above. Conceptually, the program works as follows: at the top level, all processors simulate the world in discrete time steps of length determined by the velocity and acceleration of the fish. Each time step breaks into synchronous phases for computation of forces, movement of fish, and collection of statistics, as shown in Figure 2.1. The three phases compose the bulk synchronous layer of the program. Within each phase, the code operates sequentially on the global address space (each processor looks at each fish). The program, though short, exemplifies the programming model used in many larger parallel programs.

2.2 Illustration of Use

The code in Appendix A contains annotations describing two bugs that are to be found in this section. One of the bugs we introduced purposefully, having found it in another program using Mantis; the other bug we introduced accidentally while transforming the code into a more legible format.

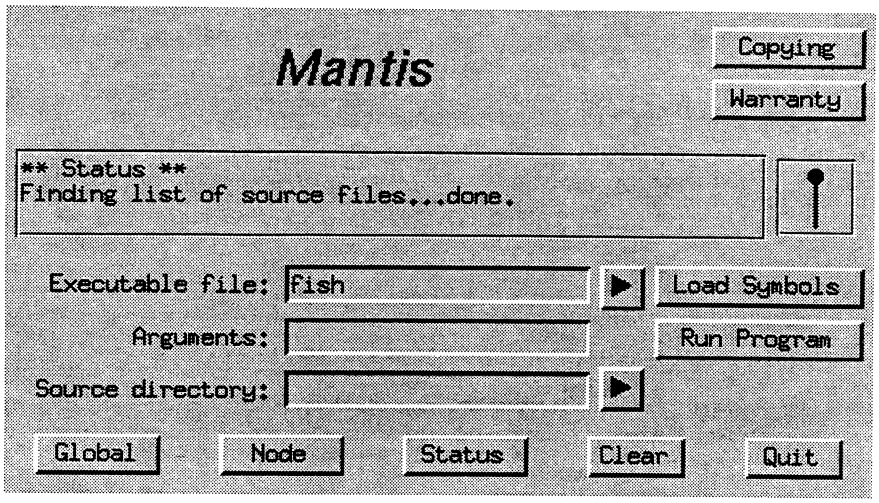


Figure 2.2: Main window. Symbols for the `fish` program have just been loaded.

2.2.1 Finding a simple bug

We compile and run the program on the CM-5, and the program immediately encounters a bus error and dumps core. To find the bug, we start Mantis, intending to run the program again.[1] After a brief disclaimer dialog, the main window appears, as shown in Figure 2.2. The main window controls high level program selection and execution, access to other windows, and the default location of source files.

The first step towards finding the bug is to locate the bus error, so we run the program by pressing the `Run Program` button and then open the status window with another mouse click. Starting a program on a CM-5 takes a noticeable amount of time, and Mantis indicates the waiting period to the user through the status area and inverted pendulum icon. The pendulum rocks back and forth until Mantis can accept another command. The user interface retains full functionality during the waiting period, but any action that requires the attention of the debugger process is stacked for later execution.

The program starts again, and the expected error occurs immediately. We detect the error with the status window shown in Figure 2.3. The green squares (the darker squares in the black and white version) represent nodes that are running (perhaps waiting for a reply from another node), while the yellow squares represent nodes in error. Nodes that stop at breakpoints or that halted by the programmer are displayed in blue.

Picking one of the problematic nodes at random, we click on the square in the status window to create the node window displayed in Figure 2.4. The signal section confirms

[1] Post mortem debugging is not yet available, primarily due to lack of complete information in the CM-5 core files.

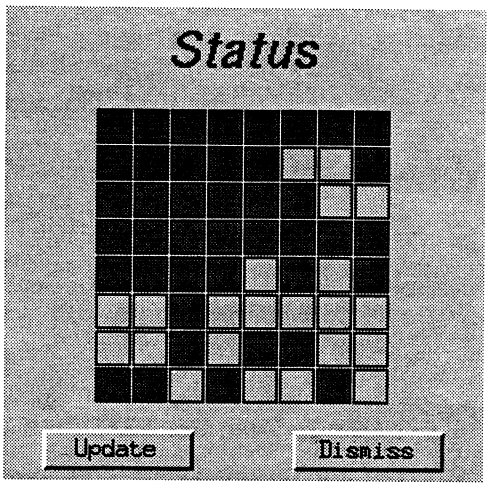


Figure 2.3: Status window. Many nodes are still running (green/dark) and many others have errors (yellow/light).

our belief that the bus error encountered from the command line caused the error, and the stack section shows that the node stopped in `all_compute_force`. Mantis focuses the source display section on the top stack frame and highlights the line that caused the bus error in black. In addition to these features, the node window gives access to the various data display windows and controls program execution on a per node basis, supporting the common single-node viewpoint inside of synchronous blocks.

Looking briefly at the highlighted source line (line 164 in Figure 2.4), we decide to investigate the variables used. We select the expression "local_fish" by pointing and clicking the left mouse button on a variable occurrence in the source display and evaluate the expression by pressing the right button. The resulting value returns in the evaluation window shown in Figure 2.5. This window provides the main interface for examining state and verifying invariants by allowing the programmer to evaluate arbitrary[2] expressions in the context of a stopped program. Evaluation includes support for Split-C's extensions to C, such as the global address space and spread arrays. The section at the bottom of the window allows the user to enter a new value for a given variable.

We learn from the window that the value of "local_fish" has not been initialized (see lines 150-151 in Appendix A). After adding the initialization, we recompile the program, hoping to encounter no further problems.

[2]Some restrictions with regard to function calls and macros exist, as discussed below.

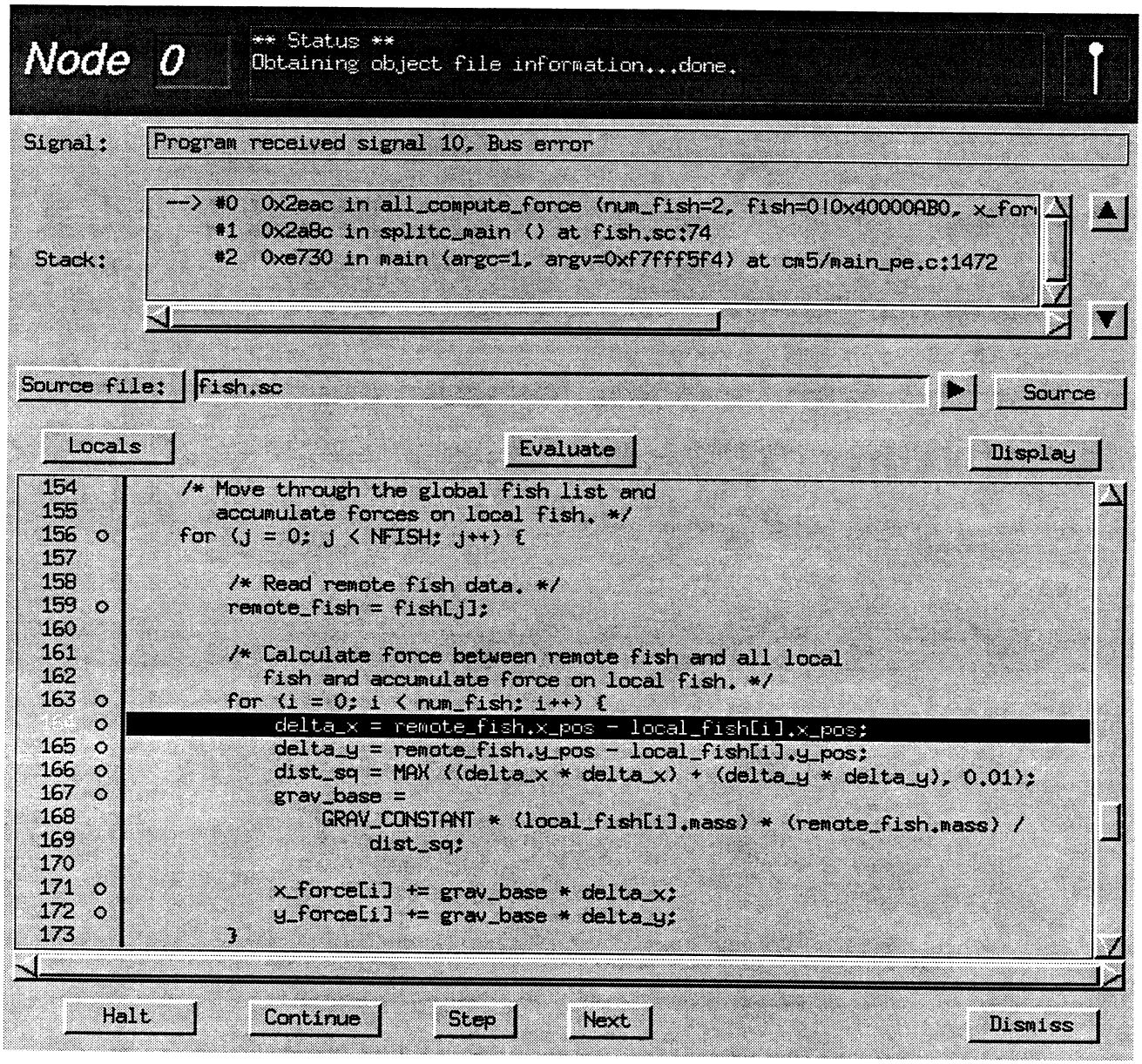


Figure 2.4: Node window. Node number 46 encountered a bus error in `all_compute_force` at the line highlighted in black.

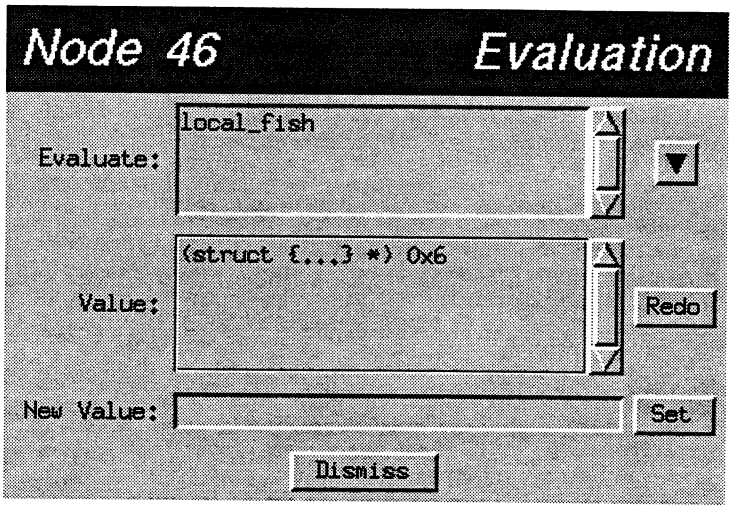


Figure 2.5: Evaluation window. Examining the expression "local_fish" reveals the cause of the problem.

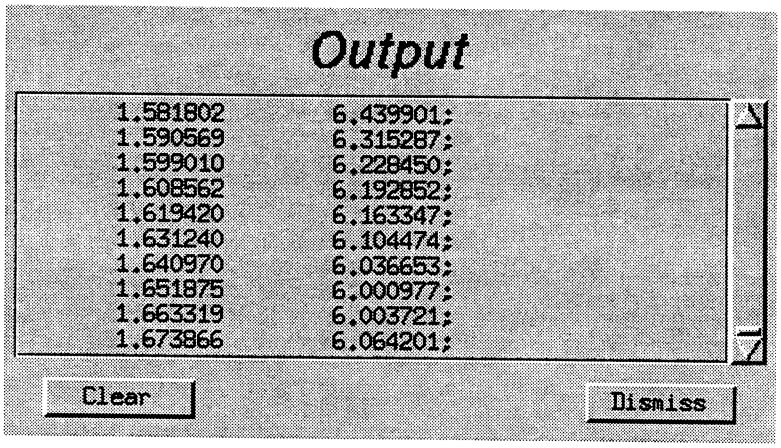


Figure 2.6: Output window. The partially debugged program hangs before completion.

2.2.2 Locating a more subtle bug

Alas, the program hangs after finishing only about a quarter of the time steps. We start up Mantis again and run the program. The output from the program begins to pour into the output window shown in Figure 2.6, but eventually stalls. The status window shows that all processors are running.

Using the [Node] button in the main window (Figure 2.2), we create a node window for node 0, as shown in Figure 2.7. By clicking on the [Halt] button, we halt the processor. The processor stops inside of a synchronization point (a `barrier` call), but we see that the stack frame just below the barrier is `splitc_main`. We click the mouse on the line containing `splitc_main`, and Mantis automatically displays the relevant section of code in the source display area, as shown in the figure. The source line of the `barrier` call is highlighted in black. Using the node number entry box in the upper left corner of the node window to move between nodes, we halt and examine a few other nodes and find that each stopped at the same barrier.

At this point, we begin to hypothesize about the problem: the chance that we happened to stop all or even most of the nodes waiting at the barrier is slim unless at least one other node is not at a barrier. By letting the processors run for a bit and stopping them again, we can make that chance arbitrarily small. The problem then becomes to determine why some processors do not reach the barrier. To get a better picture of the state on each node, we open the local variables window appearing in Figure 2.8 by pressing the [Locals] button. The window shows the values of all variables local to the frame selected in the corresponding node window. We see that the time `t` agrees with the last value shown in the output window, as we expect since node 0 performs the print statements. When we shift to another node, however, the time no longer agrees—somehow the processors broke the programmer's concept of bulk synchronous, equal-length time steps. We assume that a subset of nodes reached `T_FINAL` and left the main loop, causing the remaining nodes to wait indefinitely at the barrier.

To verify our hypothesis, we set breakpoints on two nodes at the point in `splitc_main` where `t` is updated (line number 70). For each node, we open a display window with the expression "delta_t," as shown in Figures 2.9 and 2.10. The display window is similar to the evaluation window except that it evaluates the expression automatically whenever the processor stops or a new frame is selected. We want to compare the values of `delta_t` between processors for each bulk synchronous step. Using a display window for each node, we need only press the [Continue] button in each node window between comparisons. As early as the second step, we note that the times diverge, as shown in the figures.

We have verified our hypothesis about the nodes becoming unsynchronized, but we have yet to understand how the failure occurs. To understand the process, we move upwards for a brief period and consider the bulk synchronous model. We want to stop all processors after

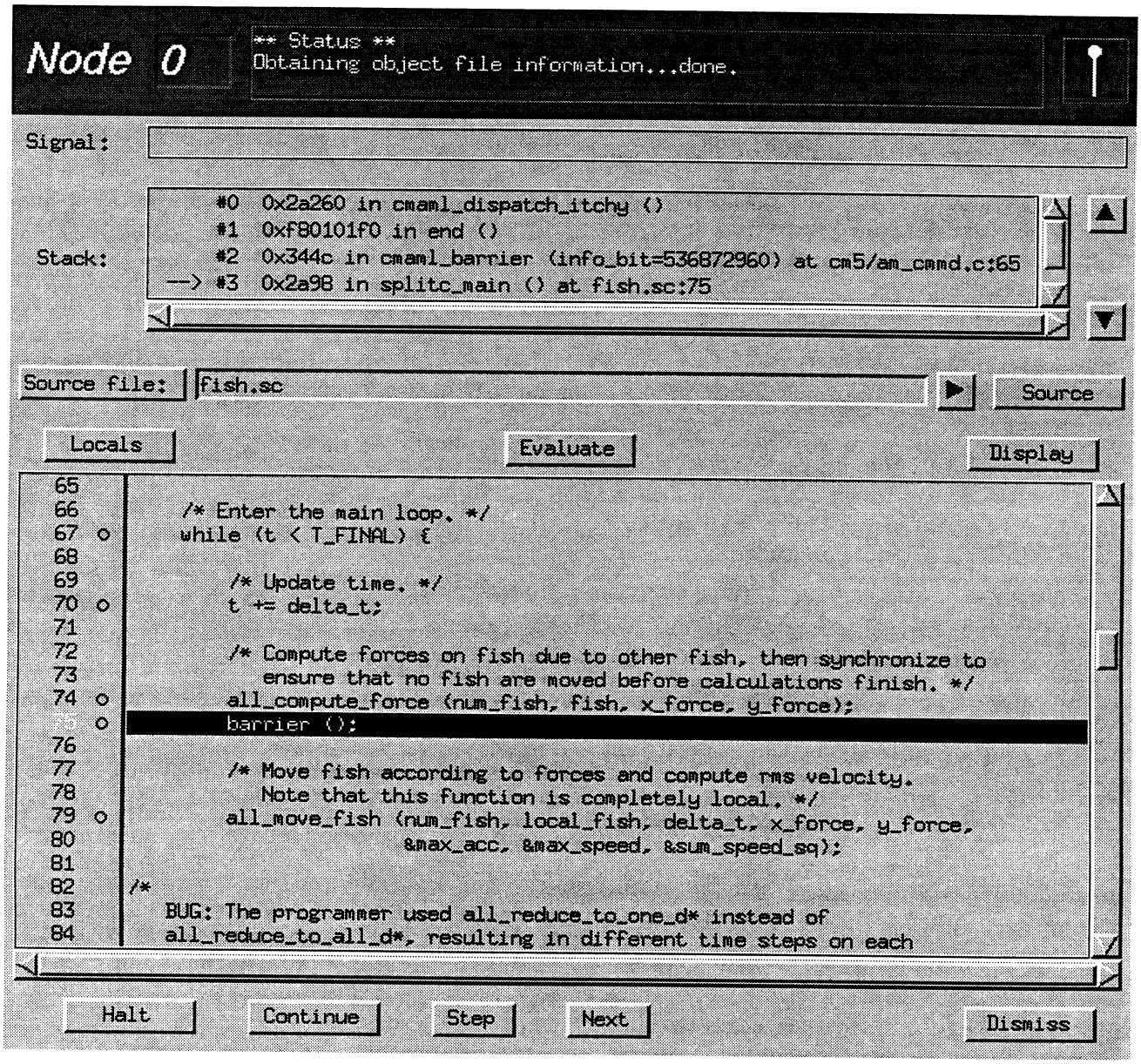


Figure 2.7: Node window. Node 0 hung in a barrier. The programmer moved up the stack to `splitc_main`, and the `barrier` call is highlighted in black.

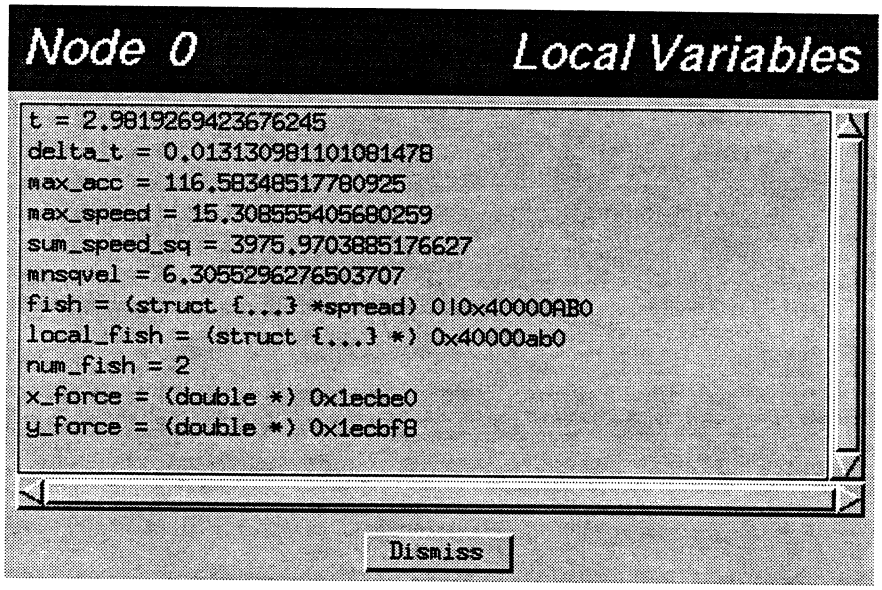


Figure 2.8: Local variables window. The variables shown correspond to the `splitc_main` stack frame chosen in the node window.

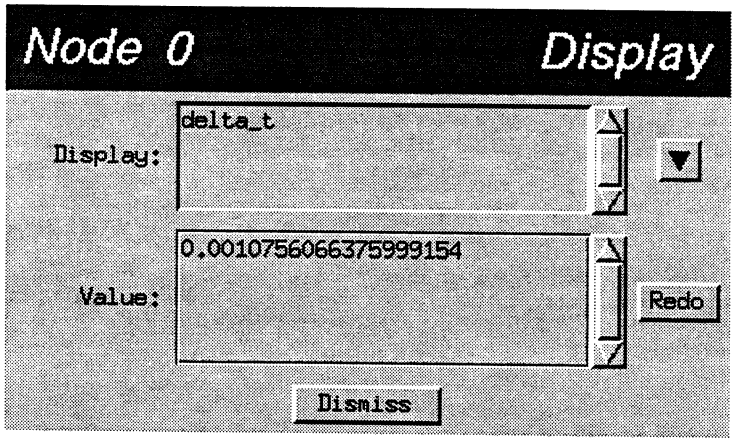


Figure 2.9: Display window. The expression "delta_t" is evaluated on node 0 each time the processor stops or the programmer changes the frame.

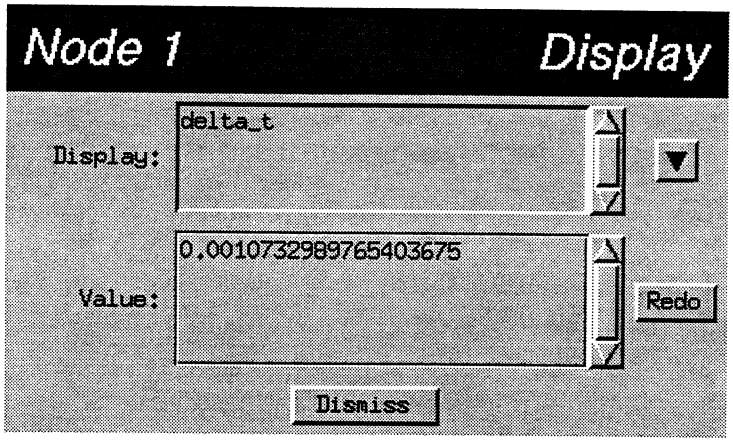


Figure 2.10: Another display window. The expression "delta_t" is evaluated on node 1.

they complete the first few phases. In particular, we want to stop them all just before they calculate the value of `delta_t` for the next time step. We click on the `Global` button in the main window to get the window shown in Figure 2.11.

The global window supports the bulk synchronous view of the Split-C program by allowing the user to toggle breakpoints and to start and stop all nodes simultaneously. We set the desired breakpoint, as shown by the solid dot at line 99 in the figure, and restart the program. Once all processors stop, we pick two processors and examine the quantities used to calculate `delta_t`. Finding that both `max_speed` and `max_acc` differ, we look up a few lines and realize that the calls to `all_reduce_to_one_dmax` returned different values on the two nodes, and we find the bug: we should use `all_reduce_to_all_dmax`, which returns the reduced value to all processors (see lines 83-86 in Appendix A). We make the changes and recompile the program, which now runs as we expect.

2.3 Summary of Features

This section summarizes the features of the Mantis debugger by topic, referring frequently back to the figures used to illustrate the example in Section 2.2.

2.3.1 Interface highlights

We attempted to make Mantis easy to use by providing an interface that adheres to standards when possible and utilizes common methods when no standards exist. The following list exemplifies the features of Mantis that appear in many known interfaces:

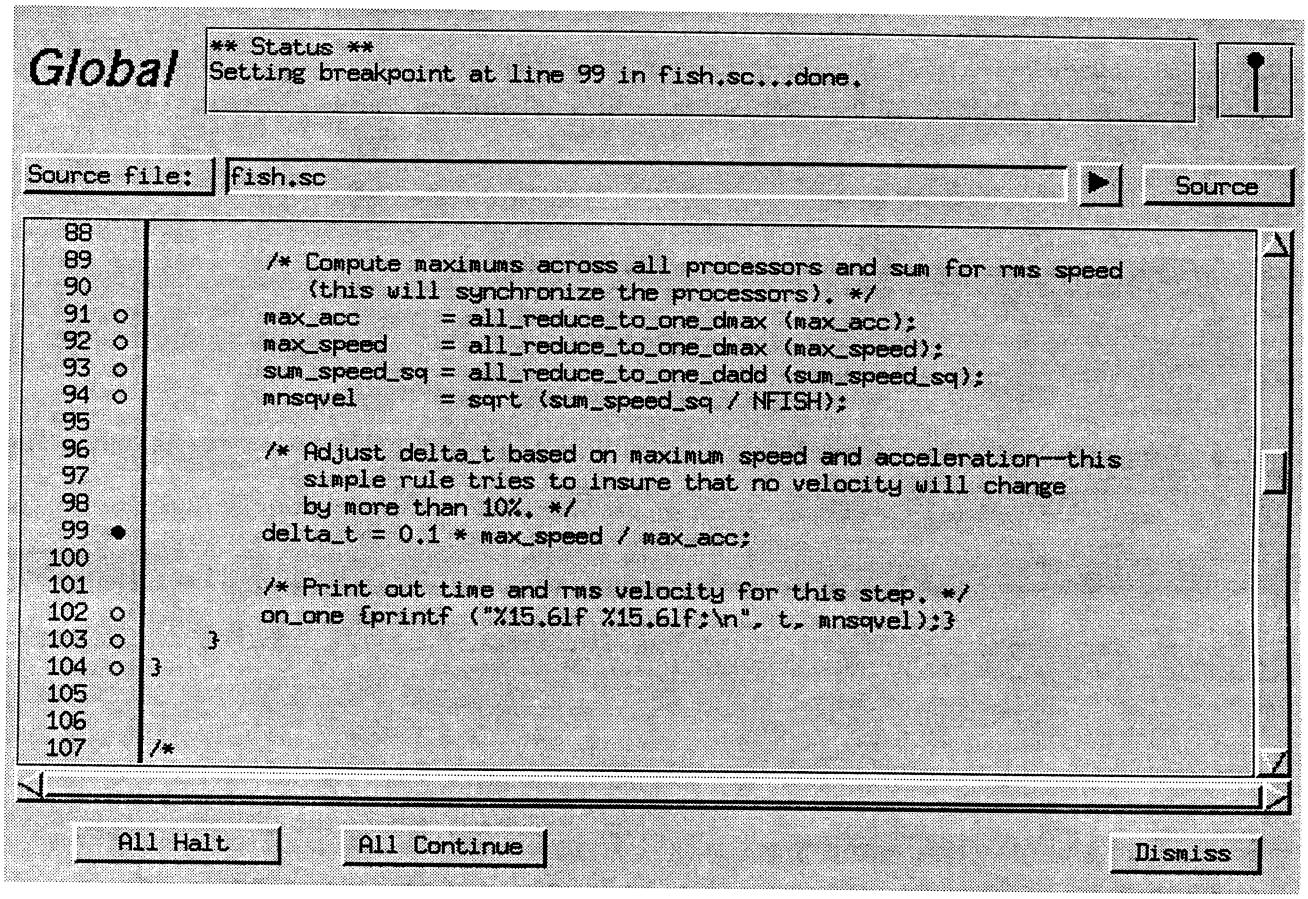


Figure 2.11: Global window. The programmer set a breakpoint at line 99 in `splitc_main`.

- Text editing supports a number of standards:
  - most commonly-used GNU controls (i.e., the `emacs` controls),
  - X cut and paste: the left button highlights text and the middle button inserts highlighted text, and
  - PC cut and paste: the left button highlights text, **CTRL-X** cuts text, and **CTRL-V** inserts cut text.
- Menus use the left mouse button for normal use, the middle button for tearoff (persistent) menus.
- Text selection and control buttons all use the left mouse button.
- The **Tab** key moves between entry boxes.
- The **Return** key executes an appropriate action within entry boxes; a control button to the right of the entry executes the same action.
- Actions that cause implicit changes in state (e.g., those that kill the process being debugged) request user verification before proceeding.
- All windows provide a control button, located in the lower right corner, used to dismiss the window.

The debugger process handles requests sequentially and without overlap, but the Mantis interface interacts asynchronously with the debugger and enqueues requests that the debugger is not prepared to handle immediately.

A status section in each major window (see Figures 2.2, 2.4, and 2.11) relays the current debugger status to the user. The status section provides feedback information on commands and errors, while an inverted pendulum to the right rocks back and forth whenever a command is in progress. Note that not all interface activity requires interaction with the debugger.

Mantis attempts to make the process of finding source and executable files as simple as possible. Source file information taken from the executable symbol table and appears as a menu of source files.[3] A control button immediately to the right of file selection entry boxes, marked with an arrowhead pointing right, allows the user to select files via a special dialog. The user can traverse directories and choose files in the dialog by double-clicking with the mouse.

[3]Files in the Split-C library are stripped out of the menu, sometimes resulting in user code with a library file name not appearing in the menu. We want to give a list of header files as well, but no information on header files is available from the executable.

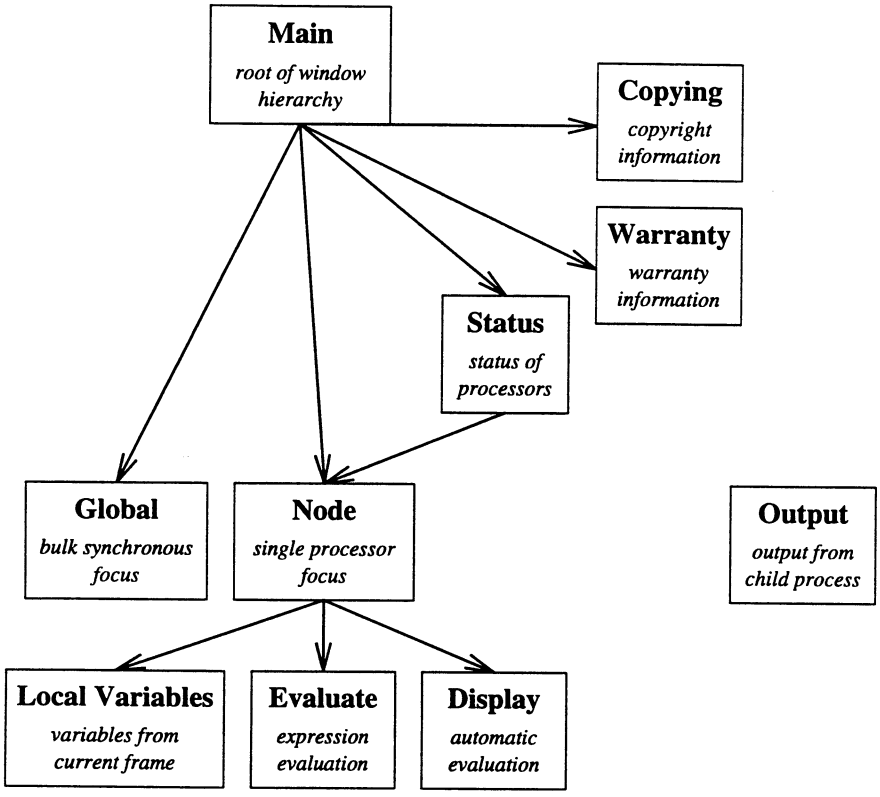


Figure 2.12: Window hierarchy. The output window appears automatically when the user's program has output.

2.3.2 Window management

The user manages Mantis windows using the hierarchy depicted in Figure 2.12. Each box in the figure represents a Mantis window. Arrows represent creation: the user can create the window at the end of the arrow by an operation in the window at the start of the arrow. The output window cannot be created by the user, but instead appears automatically when the program being debugged has output.

The main window, shown in Figure 2.2 controls access to the other major windows. The hierarchy of Figure 2.12 has roughly the same topology as the main window. The right edge of the main window holds a pair of buttons that provide information on copying and warranties according to the Free Software Foundation policies. A row of buttons along the bottom of the main window divides into two groups. The left group manages creation of other windows. Each of the three buttons generates a new window, corresponding to Figures 2.3, 2.4, and 2.11. We describe the global and node windows in later sections, and discuss the status window in the next section. The right group contains the `Quit` button and a button marked `Clear` that discards all information and closes all windows.

State	Color	Pattern
No Process	Grey	Shaded
Running	Green	Black
Halted	Blue	White
Error	Yellow	Diagonal Hash

Table 2.1: Mantis Status Window Colors and Patterns

2.3.3 Parallel process control

Debugging a process consists of selecting the executable and reading the symbolic information for the program, setting any initial breakpoints, and then choosing command line arguments and starting the program. The first and last of these operations are accomplished directly via the Mantis main window shown in Figure 2.2, while setting breakpoints typically involves browsing through the source files (see Section 2.3.4).

The upper two entry boxes in the main window allow selection of the executable and command line arguments, while the third provides a mechanism for locating source files if insufficient information appears in the symbol table of the executable.

The status window (see Figure 2.3) presents a graphical display of the current state of all processors. The processors appear as a two dimensional mesh of squares in a pattern that maximizes the size of each processor within the window size chosen by the user.[4] Table 2.1 summarizes the colors and patterns used for each possible state (patterns are used only when color is unavailable). Clicking on a processor's square brings up a node window corresponding to that processor. Thus, the user can detect errors visually and investigate via the mouse.

Both the node window (Figure 2.4) and the global window (Figure 2.11) provide controls to halt and continue processors. In either window type, a pair of buttons appears to the left, just below the source display area. In the global window, the buttons stop and start all processors, while for node windows, the buttons control only the processor being examined with the window. The node window also has buttons for stepping the associated processor through a line of code. The `Step` button continues execution until the program reaches the next line of code or enters a new procedure. The `Next` button skips over any procedure calls and stops only when the processor reaches the next line of code.

The CM-5 operating system provides no single-step mechanism for child processes, so the debugger must handle steps as a sequence of single-instruction breakpoints. Combining this requirement with the possible need for synchronization with another processor, which may be halted, we find that stepping can cause deadlock. To allow the user to override the possibility of deadlock, the pendulum pops up into button form when stepping occurs

[4]The aspect ratio may therefore change when the window is resized.

(instructions also appear in the status area). If the user presses the button, the processor halts immediately, whether or not the step has finished.

Output from the process being debugged appears in a separate window (see Figure 2.6). Both `stdout` and `stderr` are channeled into the window, which appears automatically. Two buttons at the bottom allow the user to discard the data in the window and to dismiss the window completely. In addition to output, notification that the process being debugged has exited or terminated also appears here.

2.3.4 Source browsing

Source browsing occurs through two Mantis windows: the node window (shown in Figure 2.4), which focuses on one processor, and the global window (shown in Figure 2.11), which allows interaction with all processors simultaneously. All sources appear in the large rectangular region in the center of the windows.

A line of controls just above the source display section manages access to source files and functions. A menu of selection methods on the right, appearing as `Source` in Figure 2.4, displays the method in use and determines the meaning of the entry box and the button to the left. The entry box accepts names of source files or functions, depending on the selection method chosen. The choices for selection method include:

Source indicates that Mantis should view the contents of the entry box as a source file. The button to the left of the entry, marked `Source file:` in Figure 2.4, allows the user to choose from an alphabetized menu of source files for the program.

Function indicates that Mantis should view the contents of the entry box as a function. The button to the left of the entry, marked `Function:`, allows the user to choose from a list of functions examined recently (the menu is sorted in least recently used order).

Assembly is equivalent to the **Function** method except that the source display area shows assembly code for the function instead of the original source.

The source display area splits into two sections. The left section displays line numbers and breakpoints. Empty circles denote possible breakpoints, and filled circles denote existing breakpoints. Source lines that generated no executable code have no breakpoint symbol. The breakpoint section ignores horizontal scrolling. The right section of the source display shows a portion of the source file or function. The middle mouse button toggles breakpoints on and off when the associated processor(s) is stopped.

2.3.5 Individual nodes and state

The Mantis node window provides a rich interface to individual processors. At the top of each node window (shown in Figure 2.4), an entry box shows the processor number and allows the user to examine different processors by changing the number and pressing **Return**. All status, stack, and display information corresponding to a node window changes automatically when the user chooses a new processor. Node numbers run from zero to one less than the number of processors in the machine, as they do in Split-C. The status section of the window is color- or pattern-coded to provide easy visual identification between the various windows corresponding to a given node window.

Directly below the status section of the node window is a region that displays signal and stack information when the processor halts. If a signal (e.g., a segmentation fault) caused the process to halt, that information appears in the area marked "Signal." The stack section displays stack frames, which the user can select by pointing and clicking or can traverse up or down, one at a time, using a pair of buttons to the right. When no information is available, the stack motion buttons are disabled (they turn grey). When the user moves to a new stack frame, the source display area presents the source code corresponding to the new frame, complete with a highlighted line where the processor stopped, as shown in Figures 2.4 and 2.7.

2.3.6 Data display and entry

The user can examine state in Mantis in several ways. The most commonly used method requires only two clicks of the mouse in the source display area. Pressing the left button highlights a variable or expression. Mantis tries to select text intelligently, highlighting only a variable name on the first click and expanding the highlighting on the second and subsequent clicks of the mouse. The user can specify an exact portion of the text by dragging the mouse with the left button held down. Once the expression is highlighted, pressing the right mouse button creates the evaluation window (shown in Figure 2.5) and evaluates the expression.

The evaluation window provides a fairly standard interface from the PC world for evaluating expressions and changing variable values. The expression to be evaluated is entered in the top entry box and the value is returned in the middle box. A menu button to the right of the entry box gives a list of recently evaluated expressions. The user can cycle through a small set of expressions by selecting each from the menu. The bottom entry allows the user to change the value of an expression. Errors in evaluation appear as feedback in both the node window and the evaluation window, allowing the user to focus on either.

In addition to the usual method for checking values, Mantis provides several others. A row of buttons to manage data display sits just above the source display area (see Figure 2.4). The

button layout matches that shown in the window hierarchy of Figure 2.12. The `Evaluate` button resides in the middle and is equivalent to pressing the right mouse button in the source display region. The `Display` button to the right creates the window shown in Figure 2.9, like the evaluation window except that Mantis evaluates the expression automatically when the processor halts or the stack frame changes. Because the user may want to maintain several displayed expressions, Mantis makes the display window as small as possible by removing the section used to change values. Values must be changed via the evaluation window instead. Finally, the `Locals` button to the left creates a window that displays variables local to the selected stack frame. Since local variables depend on the stack frame, Mantis updates the information automatically whenever the frame changes. Notice that Mantis marks data display windows across the top with the color and processor number of the corresponding node window. We also group the windows with the node window so that iconifying the node window also iconifies the subwindows.[5]

Because Split-C lacks a standard output format for global and spread pointer values, we created one for Mantis. Naturally, Mantis accepts both the format and standard Split-C constructs such as `toglobal` and `tolocal` on input. Any global or spread pointer output appears as follows:

```
processor!address
```

The binary global pointer creation operator is left associative and has precedence between arithmetic operators and logical operators. Hence

```
a ! b ! c  is the same as  (a ! b) ! c
a + b ! c  is the same as  (a + b) ! c
a == b ! c is the same as  a == (b ! c)
```

The operator evaluates the left argument for processor portion if possible, using the right value for the address. Thus, one can combine two global pointers and form a third using the processor of one and the address of the other. The type of the result depends on the type of the second argument. If the argument is a type of pointer, the result is a global pointer to the same type. Otherwise, the result is a global pointer to type `void`.

Creation of global and spread pointers mirrors the language. The address of a spread array section, for example, is a spread pointer if the section consists of one or more of the blocks on each processor, and is a global pointer if the section is part of a block. For example, if the spread array `a` is declared as `int a[PROCS*2]::[2]`, then the expression `&a[1]` has type `int (*spread)[2]`, while the expression `&a[1][1]` has type `int *global`. Casting to a global or spread pointer (or spread array) uses the processor associated with the window. Also, except in the case of pointer arithmetic, casting or creating a NULL pointer (a pointer with local address 0) discards any processor component and results in the NULL pointer 0!0.

[5]Not all window managers support window groups, unfortunately.

Local pointers are dereferenced on the processor associated with the window, regardless of where the pointer was obtained. For example, dereferencing a global pointer to a local pointer to an integer in an evaluation window for processor `n` returns the value at the address of the local pointer in the address space of processor `n`, just as occurs in Split-C.

Finally, since dynamically allocated arrays appear often in Split-C, we describe the notation used to cast a pointer to an array and to set the values of a dynamic array. To print a fixed number `N` of elements beginning at a pointer `p` to type `my_type`, use the form:

```
(my_type [N])*p
```

The result is an array of elements enclosed in braces. Changes are made by typing in a new array, also enclosed in braces, and pressing **Return** (or pressing the Set button). Note also that we inherit from `gdb` an abbreviation for the fairly common cast-and-dereference occurrence. If, for example, we have a void pointer `ptr` that points to something that we want to print as type `my_type`, we normally use:

```
*(my_type *)ptr
```

But we can abbreviate the expression using the following:

```
{my_type}ptr
```

Chapter 3

Issues in Parallel Debugging

In this chapter, we examine a range of issues that arise in the design of a parallel debugger and affect tradeoffs in other areas of parallel systems. We approach each problem generally, then focus on the requirements for Split-C and how Mantis responds to those needs.

3.1 Problem Localization

Debugging begins with problem localization. Sequential bugs appearing in parallel programs require little more than a fast means of finding the processor and the section of code, but inherently parallel bugs are often more subtle. Tools to compile data from across processors into a compact form help to reduce the amount of information presented and the time required to locate a bug.

Bugs of the second type discussed in Chapter 2, for example, are not atypical during the development of a Split-C program. Contrary to the programmer's expectations, processors pass beyond the bounds of a bulk synchronous block and cause synchronization to fail. The source of the problem appears readily when comparing stack traces across processors, but the user cannot be expected to compare a large number of traces by hand. Instead, the debugger can compare the traces on request and highlight the differences, directing the user quickly and accurately to the problem, as shown in Figure 3.1.

The current version of Mantis provides only limited facilities for machine-wide access, but we plan to add useful mechanisms like the stack tool to future versions.

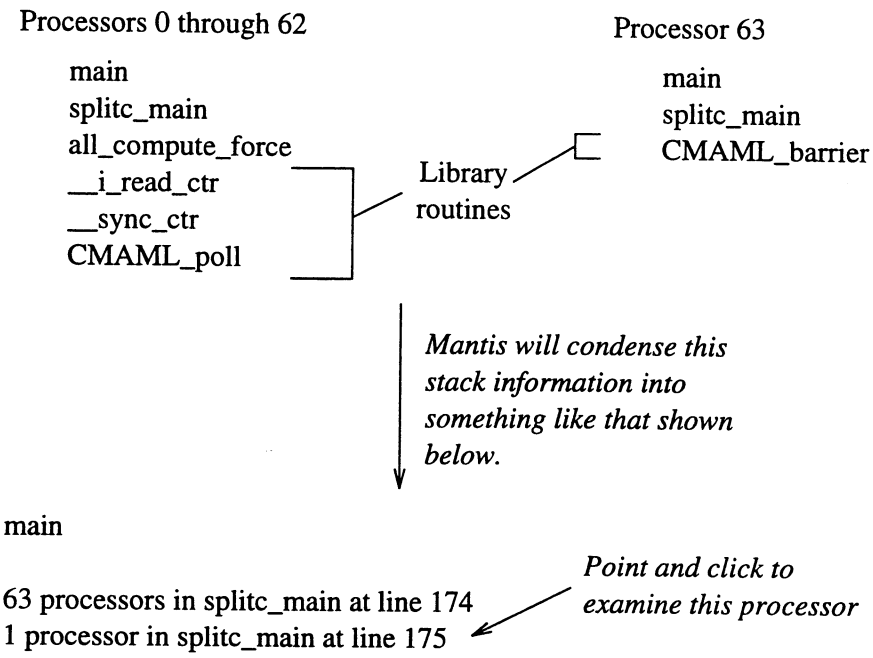


Figure 3.1: Stack trace overview. The debugger displays only differences in stack frames, maintaining the content but reducing the volume of information given to the user.

3.2 Breakpoints

In sequential languages, a breakpoint stops all processing instantaneously at some point in the program, freezing the state of the program to allow inspection and alteration. The semantics relies on the single thread model of control. Once the model is violated, providing the same semantics for breakpoints becomes difficult or impossible. Even if the programmer still envisions only a single control stream, mapping from the multi-threaded reality of the program to the single-threaded understanding of the programmer may prove impossible.

Consider a data parallel language in which the programmer has written a `for` loop over the rows of a distributed matrix—each step of the loop processes one row. Assume that, for optimization reasons, the compiler reverses the loop order and splits the work across four processors. What happens if the programmer wants to examine the state of the program after a few rows have been processed? Figure 3.2 shows a typical program state: elements in the shaded region have been processed, while unshaded elements remain unprocessed. Figure 3.2a shows the programmer's understanding, in which the loop progresses from one row to the next. Figure 3.2b shows the actual processing pattern, in which four separate threads work asynchronously through disjoint columns of the matrix. How can the debugger reconcile the state of the program with the user's expectations?

The answer is that it cannot, but the example is slightly unfair in that not only did we

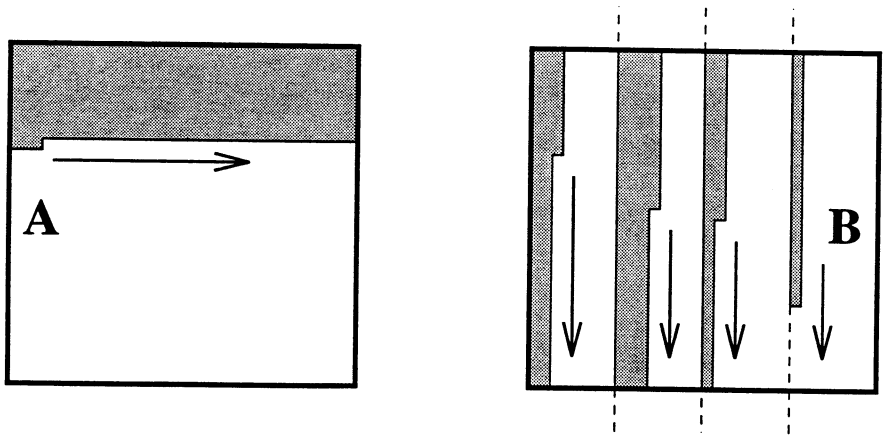


Figure 3.2: Mismatched task completion. The programmer's conception of task completion appears on the left (a), and actual task completion appears on the right (b).

allow multiple threads, but we also allowed the compiler to transform the program through loop reordering. However, note that either change results in a user recognizable state only before and after the `for` loop. Further, loop reordering is a typical transformation required for optimizing programs on distributed memory machines because of data locality constraints.

In general, a tradeoff exists between the ability of the compiler to optimize the code (including optimizing by parallelization) and the ability of the programmer to debug the program. As optimization scope narrows, the granularity of steps between states recognizable to the user similarly decreases. While prohibiting compiler optimizations may be feasible for debugging sequential programs, eliminating optimizations for a parallel program often results in unbearably long execution times and might affect numerical results as well. A better option is to add the concept of multiple threads into the language.

Once the programmer recognizes the existence of multiple threads, through either explicit language constructs like `forall` or the programming paradigm itself, as with Split-C, the debugger's task is more straightforward. Since threads only synchronize or interact with each other periodically, no exact relationship exists between the PC's of distinct processors. Instead, the programmer provides a set of larger steps, the bulk synchronous blocks, and each processor may be found anywhere within a block. In particular, since the relationship is non-deterministic, knowing the precise position of all processors when one processor hits a breakpoint is not interesting.[1]

When the programmer rather than the language defines the bounds of the bulk synchronous blocks and the interactions between processors, problems can arise as differences between the programmer's conception of the blocks and the blocks that exist in the program.

[1]Obtaining such knowledge is also impossible, since the breakpoint signal cannot be instantaneously propagated to other processors.

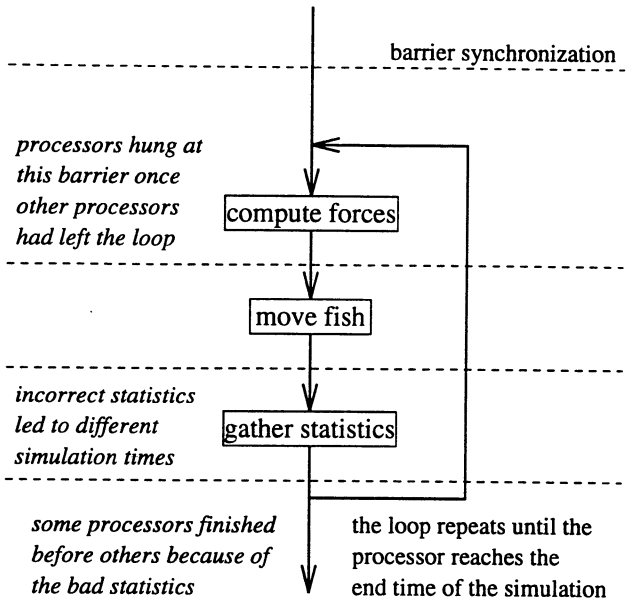


Figure 3.3: Bulk synchronous block diagram. The italicized text explains how the bug violated the block structure.

The second bug found in Chapter 2 was of this type. Figure 3.3 shows the set of bulk synchronous blocks that the programmer designed and how the program failed. The bug, due to an error in the statistics gathering section, resulted in different numbers of repetitions of the loop for different processors. Instead of all processors being inside the block for computation of forces, some processors passed off the bottom of the figure, causing the remaining processors to hang at the barrier synchronization. Another common problem arises when the programmer fails to properly separate the program, allowing processors to proceed into what should be a new bulk synchronous block before other processors complete the current block and leading to a race condition (see Section 3.4). In our experiences with users developing Split-C programs, these two problems have appeared quite frequently.

Continuing our discussion of the effects of programming models on the meaning of breakpoints, we find an even more difficult case when tasks (threads of control) are created dynamically during program execution, as with languages like Id [4] and Multilisp [14]. The first hurdle in this case is to decide when a breakpoint should fire. Possibilities include the following:

- Any processor that executes any instance of the task halts.
- If a specific processor executes any instance of the task, it halts.
- Any processor that executes a specific instance of a task halts.

The first two can be implemented in a straightforward manner, but the last presents a

fairly general problem: how can the debugger associate meaningful labels with each possible task? Regardless of whether or not the debugger provides the capability of halting on specific tasks, the debugger must be able to name the tasks in order to describe the set of tasks that exists when a breakpoint is reached. Also, the debugger must condense the information into a simple presentation for the user. Note that the set of tasks may not be deterministic, depending on the structure of the program synchronization and the location of the breakpoint, and that if other processors are not halted when one processor hits a breakpoint, the set may not even be static.

Consider now the problem of debugging the Split-C language, a relatively simple problem compared with some other parallel languages. There are no hidden program transformations; the Split-C compiler performs only the highly local variations typically performed by optimizing sequential compilers. The programmer knows that a fixed number of threads exist and understands how the threads interact.

Mantis allows the user to set breakpoints for individual processors or for all processors. The processor or processors must be halted before breakpoints can be added or removed. Each thread of control interacts independently with the breakpoints—one processor hitting a breakpoint has no effect on any other processor. Thus, not all processors will necessarily hit a global breakpoint simultaneously. In fact, if a processor still needs to reference data contained on a halted node before it can reach the breakpoint, that processor spins until the halted node is allowed to continue (but see Section 3.5 for a discussion of how to circumvent this phenomenon).

Breakpoints may correspond to source lines or to machine instructions; the user manages both types through the source display area described in Chapter 2. `gdb` allows the user to set conditional breakpoints, which halt the program only when an expression evaluates to true at the breakpoint, but the structure of the CM-5 version of Mantis prevents their use, as we discuss in Section 3.8.

3.3 Expression Evaluation Context

The context or addressing environment used to evaluate expressions depends strongly on the language of the program being debugged. If the language presents the programmer with a single thread model, then the context for expression evaluation should correspond to the context of the single thread. But mapping from the contexts of many processors back to the context of the virtual machine imagined by the programmer may not always be possible for reasons similar to those discussed in Section 3.2.

Again, when the language expects the programmer to recognize the existence of multiple threads of control, the choices are simpler. Given an addressing model like that of Split-C,

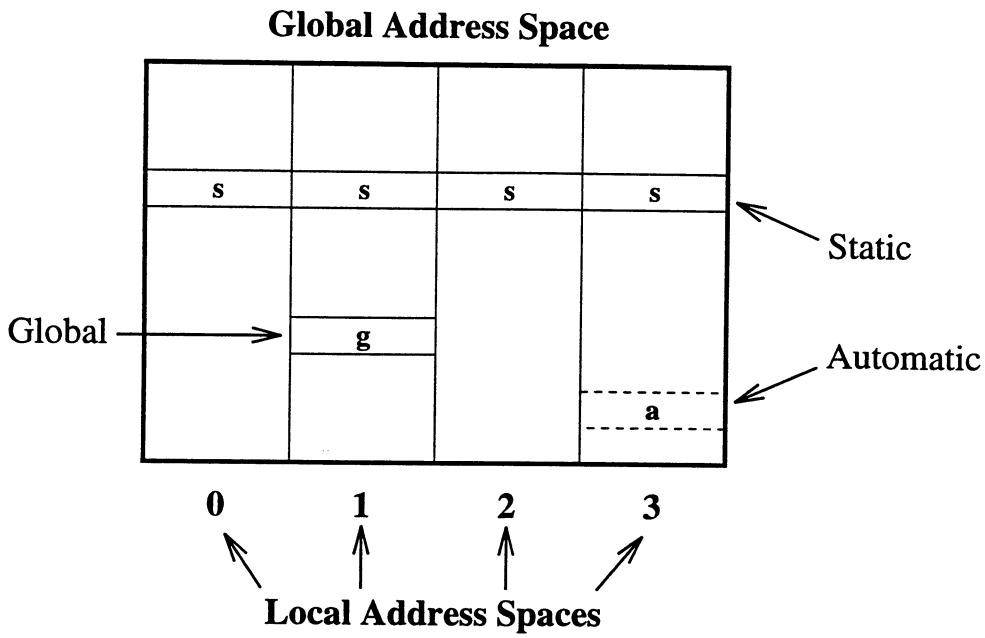


Figure 3.4: Data types in the Split-C address space. The dashed lines indicate that the automatic variable is only accessible from within a certain procedure

which allows for direct addressing of both global and local data, we can define terms for various types of data.[2] For our purposes, we need name only the three types of data shown in Figure 3.4:

global data can be named symbolically by any processor from any part of the program,

static data can be named symbolically by a single processor from any part of the program (for SPMD models, this means that each processor has its own copy), and

automatic data can be named symbolically by a single processor from within a specific instance of a specific procedure.

Any evaluation intended to correspond to a section of code must occur in the context of a single processor, but another option, one of questionable worth, is the possibility of evaluating expressions that contain only references to global data.

[2]These terms could be used equally well with schemes for dynamic translation of global addresses, but translation often requires communication with other processors and thereby causes other difficulties. We defer discussion of these problems to Section 3.5

For Mantis, we illustrate these concepts with the following program fragment:

```
1       int factorial[10 * PROCS];      /* global data */
2       int i;                          /* static data */
3
4       splitc_main ()
5       {
6           for_my_1d (i, 10 * PROCS)
7               factorial[i] = find_factorial (i + 1);
8       }
9
10      int find_factorial (int number)
11      {
12          int i, product;             /* automatic data */
13
14          product = 1;
15          for (i = 2; i <= number; i++)
16              product = product * i;
17
18          return product;
19      }
```

The program calculates the factorials of all numbers from 1 to $10 \times PROCS$. Each processor calculates ten of the factorials, calling the `find_factorial` function for each one. The program uses all three types of data. We now discuss the process of evaluating several expressions.

We can evaluate the expression "factorial[18]" without any knowledge of the addressing environment. The symbol `factorial` can only refer to the global data declared in line 1 of the program. Most expressions evaluated, however, require the debugger to understand the addressing environment.

For example, to evaluate the expression "i," the debugger must know whether or not the program halted in the `find_factorial` function. If the program stopped in `find_factorial`, the debugger should return the value of the automatic data declared in line 12. If the program stopped in `main`, the debugger should instead return the value of the static data declared in line 2. However, on a parallel machine, each thread might stop at a different point in the program, requiring different interpretations of the expression. Further, even if all processors stop at the same point in our program, the expression might still evaluate to a different value on each processor. In fact, evaluation of an expression may not be feasible on all processors. We cannot evaluate the expression "product" on a processor that stopped in `main`. The symbol `product` admits only one interpretation, but the result is automatic data (declared on line 12) and does not exist unless the processor halts in `find_factorial`.

Because most expressions evaluated contain ambiguous references or references to static or automatic data, Mantis requires the association of a processor with an expression for evaluation. The node window manages the data display windows, so Mantis uses the processor associated with the node window for expression evaluation.

In languages that allow multiple threads of control on each processor [4][14], the debugger must associate expression evaluation with single threads. The Split-C model of one thread per processor allows Mantis to make the association with processors instead.

3.4 Race Conditions

Race conditions pose one of the hardest problems in parallel debugging. Bugs of this type often arise from differences between the programmer's conception of the synchronization patterns and the actual patterns, but can also be simply a failure to fully understand the data dependencies of the algorithm.

The difficulty with race conditions is threefold: first, if the race is close, the bug may not always appear; second, any perturbation in the program—printing, tracing, halting a processor with the debugger—may resolve the race and make the bug disappear; and third, race conditions usually cause indirect crashes, making them more difficult to locate.

Perhaps most difficult to handle are those race conditions that corrupt data but do not crash the program. Results from known problems can usually be verified with the aid of careful inspection and visualization tools, but when the parallel program solves new problems, detecting incorrect results may be impossible. Some data parallel languages, notably High Performance Fortran, propose to eliminate race conditions entirely by making interprocessor activity solely the domain of the compiler. Whether the benefits of these languages outweigh the performance cost of inhibiting optimization remains to be seen. Alternatively, new programs can be subjected to the same test used with new sequential computations: write two different algorithms and compare the results.

A suspected race condition can sometimes be verified as follows. In each bulk synchronous block, halt one processor at the start of the block, allowing the other threads to run to the synchronization at the end of the block. If the data used by the halted processor in the block change, a race condition exists. If nothing changes, run the single processor through the block while keeping all other processors halted. If the results written by the halted processor in the basic block are incorrect, a race condition exists. Naturally, if in either case the processors fail to reach the correct synchronization point, the user should inspect the bulk synchronous block structure more closely.

The CM-5 version of Mantis does not provide any special tools for finding race conditions,

but in future versions we plan to support the method outlined above. The method requires that halted processors handle and respond to messages, allowing the running processors to access remote data. In the next section, we discuss the merits and drawbacks of making active messages truly active.

3.5 Active Messages

With Split-C on the CM-5, we chose to extract messages from the network via polling instead of interrupts, rendering "active messages" somewhat passive. Because the messages are not extracted from the network until the receiving processor polls or attempts to send a message, halted processors ignore the network, effectively freezing a portion of the global address space. In Split-C, only the memory corresponding to a halted processor is unaccessible, but in languages and libraries (such as Orca [2] and Tarmac [11]) that dynamically translate global data names into physical addresses, all of remote memory can freeze when any processor halts.

More modern architectures like the Cray T3D provide DMA units to automatically write data into the processor memory, making the messages truly active and moving closer to shared memory.[3] Keeping the entire address space accessible when a processor halts is not always an advantage, however.

Consider a case in which one processor writes an address into a global variable in a second processor's memory. When the message arrives, the address becomes valid. Assume that the programmer has forgotten to synchronize the use of the address with the arrival of the message, and that the second processor crashes in an attempt to use the address before the message arrives. Before the user can respond to the error, the message arrives. Should the message be handled? Certainly not—the message will change state and hinder the process of finding the bug. State must be preserved to allow the user to correctly backtrack through the code executed prior to the crash.

Both options have advantages and disadvantages, and the best choice is to give control to the user. One way to accomplish this is to add a library routine that does nothing but handle messages. The routine, which will be short and will not perturb the program, can be included automatically in all Split-C programs. By calling the routine at the request of the user, Mantis can unfreeze the address space of a halted processor.

[3]The operating system can prevent messages from being delivered to halted processes in some cases.

3.6 Procedure Calls

The ability to call program functions from within the debugger often proves useful, particularly when the program contains a large amount of state information that is otherwise difficult to search. So long as the functions operate within the local address space and do not synchronize with other processors, no problem arises. Routines that read or modify data or perform atomic operations on other processors will hang if the other processor has halted without the modifications discussed in the previous section.

The user must also avoid calling routines that modify state when a processor halts in a critical section of code. Current versions of Split-C use polling rather than interrupts or DMA to deliver messages, and messages are only received when the code communicates with another processor. Large portions of the code are thus protected by default, and the user must understand the consequences if the protection is violated using Mantis.

The debugger can allow the user to call global functions provided that the functions are called simultaneously on all nodes, but this operation can be tricky. The global communication functions in the Split-C library, for example, use static state and are not re-entrant. Even if the user halts all processors to ensure that a global function will properly synchronize, the program results may still be corrupted.

3.7 Linear Time Operations

All parallel tools suffer to some degree from lack of scalability. For tools restricted to a small number of processors, scalability does not matter, but if the tool must handle a large number of processors or be useful on any parallel platform, per-processor sequential tasks should be avoided. Ideally, all commands and operations are performed in parallel, allowing the tool to perform as well on thousands of processors as it performs on a handful of processors. If nothing else, however, the user interface must be sequential. The best tools parallelize operations to the point that for any machine size used, the slowest step is that of presenting the data. Data display should be optimized, of course, but gathering the data should be fast and independent of the number of processors.

Unfortunately, the point is complicated by scheduling issues. On the CM-5, for example, the Time Sharing Daemon gives equal time to each process regardless of status. In order to parallelize operations, the debugger must run as a parallel process (as does Node Prism, described in Section 6.1.2). Assuming that the machine is otherwise unoccupied, the debugger will be scheduled for half of the time even when idle, doubling the time needed for the program being debugged to progress.

On massively parallel machines like the CM-5, however, the sequential alternative is far worse. When we wrote Mantis, the CMOST operating system did not support debugging programs directly from the parallel processors. Instead, we had to operate through the Time Sharing Daemon on the host processor, allowing only one debugger process and making moot the idea of parallelizing operations. Certain operations in Mantis require time linear in the number of processors, primarily those associated with the global view of the program: halting and continuing all processors, setting global breakpoints when the program starts, and synchronously checking the status of the processors. Each takes on the order of a second when running on a 64-processor machine. On future platforms, Mantis will parallelize these operations.

3.8 Asynchronous Notification of Status

Sequential debuggers typically ignore the user while the child process runs, paying full attention to the child process. The debugger thus receives immediate notification of any changes in status due to breakpoints or errors and passes the status change information quickly to the user. The ability to detect status changes asynchronously without operating system support requires the attention of the debugger process, and parallel debuggers must create stub processes for each thread in order to retain the ability.

The CM-5 version of Mantis uses only a single debugging process, working directly with each thread through the CM-5 Time Sharing Daemon. By changing the interface between the debugger and the child process, Mantis forfeits the ability to detect status changes asynchronously. Instead, Mantis polls each processor periodically to detect changes. Since polling takes $\theta(P)$ time for P processors, Mantis polls in 5 second intervals, and changes in status reach the user on average 2.5 seconds after they occur. The lack of debugger stubs also prevents use of conditional breakpoints.

3.9 Summary

As discussed in this chapter, many problems arise in the development of a parallel debugger. Several of the problems depend strongly on the degree to which the programmer's conception of the program matches the compiled code. Some languages attempt to ease the programmer's task by presenting a single-threaded model and eliminating the possibility of race conditions, but debugging this type of language can be hard: in many cases the debugger cannot translate the state of the program into something recognizable by the user. Further, automatic compilers can only obtain reasonable performance for a small class of very regular problems. Explicitly parallel languages make the debugging task more straight-

forward and permit the programmer to obtain maximum performance for the program, but cannot prevent the user from introducing difficult parallel bugs such as race conditions. If the language provides for dynamic task creation, the debugger must also be able to assign meaningful names to the tasks.

The issue of handling messages on halted processors has good arguments for both sides. Not handling the messages prevents the user from exercising full control over the machine, since allowing one processor to run may require that another be allowed to run at the same time. On the other hand, handling the messages can destroy important state and confuse the user. The best option is to ignore the messages by default and to allow the user to override the default when necessary.

We also noted that procedure calls are a useful mechanism in debugging and should be included in parallel debuggers. The user must be particularly careful, however, to avoid corrupting data by calling a procedure when the processor halts in a critical section or non-re-entrant library routine.

Finally, we saw that parallel debuggers must operate in parallel. The time required for a single process to gather data or operate on P computing nodes grows as $\theta(P)$, making many operations slow unless parallelized. More importantly, however, a single-process debugger cannot monitor all threads simultaneously, greatly increasing the response time of the debugger to changes in thread status and making several useful debugging abstractions impossible.

Chapter 4

Experience with Large Software Systems

Mantis is composed of a graphical user interface process that pipes information to and from a debugger child process. The child process performs the actual debugging, handling all typical debugging tasks, while the user interface attempts to present the information in a more accessible and automatic fashion than that provided by command-line debuggers.

We wrote the user interface using the Tool Command Language (Tcl) [15][16] and X11 toolkit (Tk) [17] developed by Ousterhout, which greatly simplified the task.[1] The interface required about eight thousand lines of code, which divide equally into script code and C code.

We based the Mantis debugger on the Free Software Foundation's `gdb` debugger. `gdb` is composed of over two hundred thousand lines of code and provides a portable sequential debugging environment. Understanding and extending the `gdb` code to support parallel debugging proved fairly difficult, and complications with the CM-5 operating system made the modifications to the code extensive. Nevertheless, adding language support and parallel processes to `gdb` required much less time than that required to write a high-quality debugger from scratch. Furthermore, `gdb` eases portability problems by allowing us to incorporate our changes into future releases. Modification of an existing tool worked to great advantage with Split-C, which uses the `gcc` compiler, and we expect the same results with Mantis and `gdb`.

This chapter discusses the use of `gdb` as a starting point from which to build a parallel debugger, and in particular Mantis. We begin by listing the basic extensions required to turn a sequential debugger into one capable of supporting parallel language debugging. We then

[1] The value of Tcl's interpreted script nature for rapidly creating attractive, functional, and consistent user interfaces is hard to overestimate.

propose a debugging model intended to minimize the size and breadth of gdb modifications and consider the required coding efforts using this model. Next, we talk about the model used to build Mantis on the Thinking Machines CM-5 and how that model affected the coding process. We cover certain issues in more detail, and finally move on to discuss our experience and the tradeoffs involved in using a base platform like gdb as opposed to writing all of the code for the project.

4.1 Extensions Required for Mantis

Because Split-C is a new language, and because it explicitly recognizes the fact that the program runs on many processors simultaneously, a sequential debugger like gdb must be changed on many levels to be useful for debugging Split-C programs. Beginning with low-level issues and moving gradually upwards, the list of changes includes the following:

BFD support The debugger must understand the executable file format. gdb uses the BFD (binary file descriptor) library, which handles many common formats.

target type The debugger must recognize the machine architecture and the access methods used to read and write data to the process being debugged. gdb already has targets for sequential child processes, core files (post-mortem debugging), and debugging over serial lines.

multiple processes The parallel program consists of a large number of processes running in parallel. The debugger must provide the means to access and control all of these processes simultaneously. gdb provides no capabilities for this purpose.

language extensions The debugger must support language constructs to allow for natural output and entry of expressions, values, and other language-specific data. gdb recognizes C and several other languages, but support for Split-C abstractions must be added.

user interface The debugger must provide the user interface with data usually considered internal to the debugger, e.g., which lines in a source file correspond to actual machine instructions and which are merely comments or otherwise empty statements.

This list does not specify the form of possible implementations. Implementation decisions are instead based on the abstract debugging model chosen. This model is sometimes forced upon the debugger by the operating system, as we will see in Section 4.2.2.

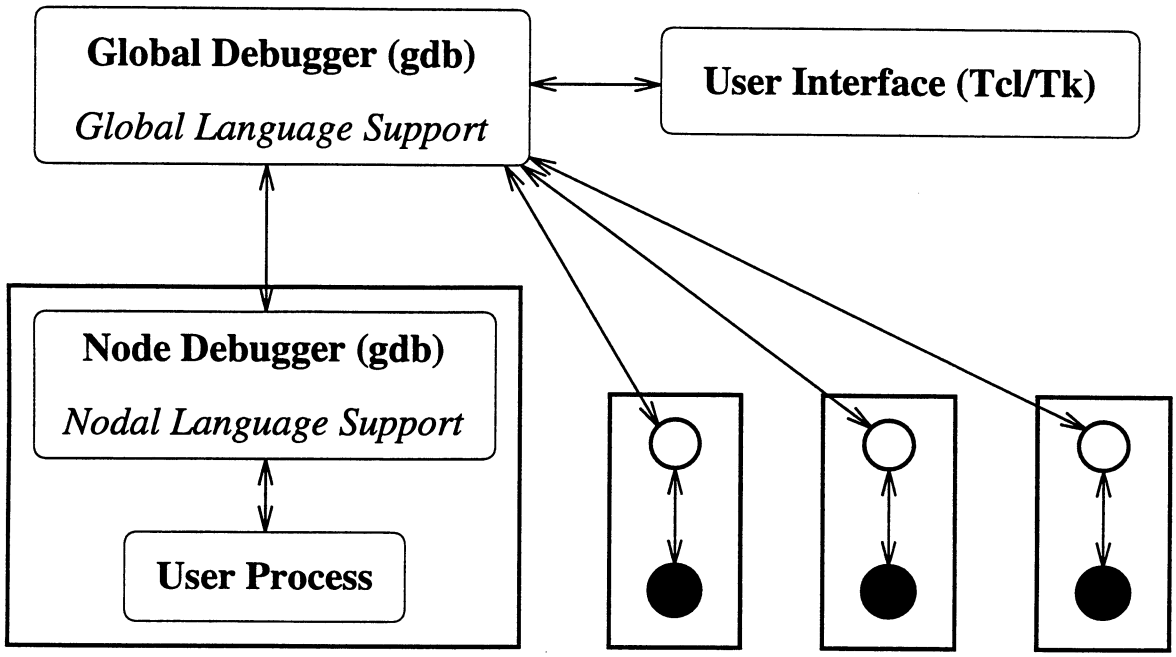


Figure 4.1: Ideal implementation structure. The node debuggers use a standard sequential interface to the child processes. The global debugger provides machine-wide functionality.

4.2 Effect of Debugger Model on Implementation

We now examine two models of debugging, one based on making a minimal set of changes to the existing sequential debugger (*i.e.*, `gdb`), and the second based on working with system software on the CM-5. We then focus on some of the specific problems of the latter model.

4.2.1 Ideal set of changes

Being a sequential debugger, `gdb` is most easily used on a one-to-one basis with the parallel processes. For each process running as part of the parallel program, we want to use a separate copy of `gdb` to monitor that process, as shown in Figure 4.1. In the figure, the bottom four boxes represent processors running a parallel program; each node also runs a debugger process that interacts directly with the user process on the node. Each node debugger also interacts with a single global debugger that gathers information from the nodes and directs user requests to appropriate nodes. The user interface process communicates only with the global debugger.

Given this model for building the debugger, consider in turn each of the general extensions to `gdb` given in Section 4.1.

First, the debugger must understand the executable file format used for the parallel program and recognize the machine architecture. For almost any network of workstations, or for any MPP built with common processors and running a fairly standard operating system software (e.g., the IBM SP-1 and SP-2 and the Meiko CS-2), `gdb` can be used without modification. But even if the current version of `gdb` does not support a particular file format or architecture, adding that capability to `gdb` is still far more productive than writing the entire debugger, since the effort indirectly provides other researchers who want to port debugging tools to the same machine with a stable, open, and supported platform for their work.

Second, we must make the debugger handle multiple processes. With the proposed model, a distinct `gdb` process monitors each process of the parallel program; all of the `gdb` processes maintain the one-on-one interface already present in the sequential version. We need only create a modified version of `gdb` to ease access to the many processes. However, since this global debugger only communicates with other `gdb` processes, it will seldom or never need to incorporate new releases of `gdb`. The node debuggers, on the other hand, might have to be merged with later releases to allow Mantis to run on an expanded set of platforms. We try to keep changes to the node debugger to a minimum, as does the model of Figure 4.1. Restricting changes to the global debugger is not as important.

Third, we must add support for the Split-C language. New data types include global pointers, spread pointers, and spread arrays. The node debugger must recognize the new types in symbolic information and cast them into appropriate C data types within the local address space. The global debugger must handle access to these new data types: the expression parser and evaluation code must translate new data types and global references into a form understood by the node debuggers, consisting of only local references and C data types. The global debugger must also answer questions about the number of processors and other parallel issues. As before, the number of changes to create the node debugger from `gdb` is minimal, and while the changes to create the global debugger are more extensive, the latter changes will not need to be combined with new releases of `gdb`.

Finally, certain information normally considered internal to `gdb` must be made available for the graphical user interface, and some output formats must be changed slightly. These modifications require only simple changes and the addition of a few new commands[2] and integrates easily with new releases of `gdb`.

[2]The changes mentioned here are also those required to generate a sequential version of Mantis. The patch file for the sequential version contains approximately one thousand lines.

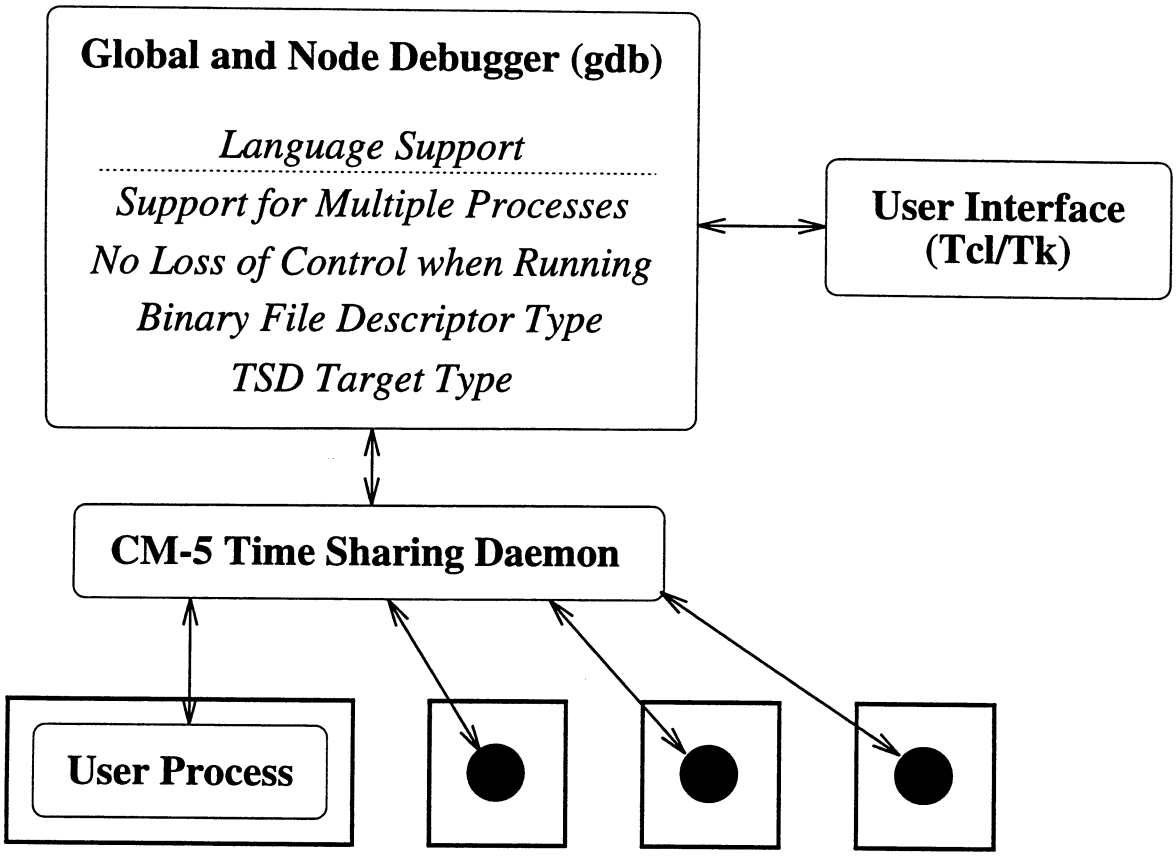


Figure 4.2: CM-5 implementation structure. A single process combines the global and node debuggers, interacting with the child processes via the Time Sharing Daemon

4.2.2 Changes required for the CM-5 version

Unlike Split-C, Mantis currently runs only on the CM-5. When we built Mantis, system software on the CM-5 forced the debugging model to vary quite dramatically from that given in Figure 4.1.[3] Under the CMOST operating system, process state information must be obtained through the Time-Sharing Daemon, which runs on the CM-5 host processor. We use a single debugger process on the host to support this model. The process communicates with the TS-Daemon to gather debugging information, as shown in Figure 4.2. As the figure shows, the user interface work remains the same, but the modifications to `gdb` are combined into a single version. In addition, the interface between `gdb` and the user process has changed, working now through the TS-Daemon.

Consider again each of the extensions listed in Section 4.1, this time in light of the CM-5 debugger model.

[3]TMC later added support for accessing other processes from the nodes for use with the Node Prism debugger.

The debugger must first understand the executable file format. CMOST uses a format similar to that used in SunOS but combines two executables, one for the host processor and one for all of the node processors, into one file. To join the programs, CMOST concatenates the SunOS binaries and adds a special structure at the end which points to the beginning of the node code in the file. Adding a binary file descriptor (BFD) to interpret the CMOST format is not inordinately difficult since a BFD already exists for SunOS, but the BFD library does consist of roughly fifty thousand lines of code that could otherwise be ignored. Also, debugging on the CM-5 requires access to the host program, but the user is interested only in the node program. Since `gdb` maintains symbols for only one executable, we must modify the code to read and manage symbols for both halves of the CMOST executable.

The debugger must also recognize the machine architecture and the access methods used to read and write data to the process being debugged. The architecture presents no difficulty, as `gdb` already contains everything necessary for the Sparc. The access methods present a much more difficult problem, however. `gdb` interacts with the program being debugged through an abstraction known as a target. Targets for sequential child processes, core files, and serial line debugging are already built into `gdb`. The program is normally only associated with a single target, but with the CM-5, we must access both the host process via the SunOS `ptrace` interface and the node processes through the TS-Daemon. Permitting the dual access requires either the use of two targets or a modification of the target structure, neither of which can be done very cleanly with `gdb`. We decided to extend the target structure to support selection between processors, a modification also necessary for switching between the various node processes. The host processor uses a distinct identification value, and the underlying routines choose access methods based on whether the process being accessed lives on the host or on a node.

Next, the debugger must provide the means to access and control the processes that make up the parallel program. Memory on the host processor does not permit a separate copy of `gdb` for each user process, so the debugger process must interact with a large number of child processes instead of the usual one-to-one relationship between the debugger and the child. Adding the capability to interact with multiple child processes to the `gdb` debugger was the most difficult modification required, and also the hardest to integrate with new releases since the changes permeate a large fraction of the `gdb` source. Further, the changes to the debugging model and problems that arise from those changes (described in the next section) are enough to convince us that the CM-5 model is not worth porting to other platforms.

Finally, the debugger must support Split-C constructs and provide certain information to the graphical user interface for presentation to the user. The required changes are essentially the same as those made under the ideal debugger model, except that all changes are made to a single version and must be integrated with new releases of `gdb`, making them more troublesome for future efforts.

4.2.3 Complications of the single process debugger model

In this section, we examine the problem of the single debugger process in more detail. Much of the effort to build Mantis from gdb dealt with extending gdb to work with multiple processes, an extension only necessary because of the restricted process access methods available on the CM-5. Although the graphical user interface relieves much of the burden that would otherwise be forced upon the user because of these changes, we could not provide complete insulation. Also, the coding required is extensive, spread throughout most of gdb, and not very portable.

The biggest problem in dealing with multiple processes in gdb, a problem that spawns many other problems, arises from gdb's handling of process control, input, and output. gdb is normally inaccessible to the user when the child process is running. gdb forwards user input to the child along with keyboard interrupt and stop signals (**CTRL-C** and **CTRL-Z** respectively) and relays program output to the user's tty. The user cannot check stack frames, evaluate expressions, or do anything else with gdb while the child process is running. gdb meanwhile waits quietly for the child process to stop, whether by hitting a breakpoint, finding an error, or receiving a user interrupt, at which point gdb immediately wakes up and interacts with the user, giving prompts and handling all input and output itself. The child process remains completely dormant during the latter phase until the user gives a command to continue.

Once gdb must manage multiple processes, the whole system breaks down. We can no longer expect that all processes be halted while the user examines any of them, nor can we allow gdb to ignore the user when any of the processes run. We can still enforce the prohibition on accessing a processor's registers or local memory while the corresponding process is running, but we must allow the user to interact with a halted process regardless of the status of other processes.

Making gdb provide continuous attention to the user requires fairly extensive changes. Prohibitions against accessing the data of a running process, previously inherent in the control strategy, must now be inserted into the code for each command. We must also provide commands for the user to switch between processes and to deal with all processes at once and must add a way for the user to halt a process. The major change in interaction management also complicates other, more subtle issues.

gdb, for example, can no longer watch the running processes to detect when a process stops. Thus, the debugger does not always have current information on the status of each process and must check the status before handling a command or notifying the user of changes. The user must ask gdb to determine whether or not a process has hit a breakpoint, for example.

Although we can afford to have the gdb process ignore most running processes, certain

operations require its full attention. Among these are the Next and Step operations, which perform one machine instruction at a time until the PC passes out of the current source line. One can imagine how long these commands might take if the gdb process polled the status of the stepping process every second. Instead, we assume that the user's attention is focused on the process being stepped for the brief time required to perform the step. Since we are dealing with parallel processes, however, we must account for the possibility that the process being stepped tries to synchronize with a halted process. In this case, the user interface can send gdb a signal to immediately halt the process.

Since we have lost the ability to asynchronously notify the user of changes in process status, we must provide a command that allows the user to poll that status.

Finally, we must extend the breakpoint abilities of gdb to allow for breakpoints that extend across multiple processes. However, we cannot allow breakpoints to be modified on running processes, since gdb inserts breakpoints when a process starts and removes breakpoints when a process stops. These restrictions must again be added into the code since the user can now issue commands while processes are running.

Another major problem with using gdb to manage multiple processes is that gdb maintains a large collection of global data specific to the child process. The data does not reside in a single place, but is instead spread throughout the code modules that make up gdb. Much of the data must be duplicated for each parallel child process. Perhaps the best solution to the problem is to change the variables into arrays (changing all declarations and references) and to perform dynamic allocation once the number of processors is known. Unfortunately, such changes require an inordinate number of modifications to the gdb code, all of which must be redone to incorporate each new release of gdb. A more manageable solution, the one we chose to implement, involves adding the storage and dynamic allocation code, but calling special procedures to swap the appropriate values into the global variables when gdb focuses on a new process. Unfortunately, the swapping can be fairly time-consuming, particularly when the debugger must switch many times between processors (*e.g.*, when gathering data on a spread array).

4.2.4 Asynchronous notification revisited

We now return to the problem of the debugger being unable to asynchronously notify the user when a process has halted. This inability leads to several problems besides the simple nuisance of having to poll the status of each process. Since several capabilities of gdb require the full attention of the debugger process, these capabilities are lost in Mantis. Conditional breakpoints, which allow the user to create a breakpoint that halts the program only when a given expression evaluates to true, cannot be used with Mantis. Nor can watch points,

which stop the process when it modifies a value at a particular memory location or locations.[4] Note also that the synchronous poll takes $\theta(P)$ time for P processors. The time required is significant when running on a large machine, and can be slow even on small machines when other parallel processes compete for time.

Remember that the ideal model, which we plan to implement on other platforms such as networks of workstations, pairs a debugger process with each user process, allowing fast asynchronous notification and making polling unnecessary.

4.3 Experience and Tradeoffs

In this section, we comment on the overall usefulness of gdb as a platform for building parallel debuggers, giving specific attention to some of the benefits and drawbacks involved.

First note that gdb is a very big program, consisting of roughly two hundred thousand lines of code counting various libraries, and when we first used the package, we had trouble deciding where to look. The Free Software Foundation tries to give good documentation, and succeeds to some extent, but the documentation appears primarily at the file level. Figuring out which gdb file contains a certain piece of information can be difficult. A central file that explains the abstractions and capabilities at a high level and points to an appropriate file for more detailed information might ease the learning process. Such a file could also contain commentary on possible changes and extensions to gdb and how they might fit in using the existing structures. Support for multiple process management, for example, goes against many of the concepts utilized in the development of gdb and therefore gives rise to many problems. gdb is not designed for certain changes, and such changes should be avoided whenever possible.

While we worked with gdb, we noticed many hooks, perhaps intended to allow people to use the gdb code by merely defining the hooks. We are unconvinced, however, that one can provide a general enough set of hooks to be useful to a person who must make significant changes to gdb.

However, despite the difficulty of dealing with such a large software package, we still feel that modifying gdb was far easier than writing our own debugger, since many areas of gdb remained unchanged and we avoided the design of many other interfaces. We also benefit from future releases—bugs in unchanged sections of the debugger will be fixed by other people, and other people will provide most of the work necessary to port our debugger to new platforms. The latter relies on our using the more natural model of debugging with

[4]Watch points are already very slow on most machines—the processor is single-stepped and an expression evaluated after each machine instruction.

gdb, of course, since the changes required for the CM-5 version are fairly extensive and for the most part unusable on other platforms.

Chapter 5

User Interface Design

The user interface is an important part of a tool because a good interface makes an average tool worthwhile and a bad interface renders a good tool worthless. While agreement on the value of specific features of a user interface is rare, we can put forth ideas upon which everyone can agree and then use those ideas to develop good interfaces.

We began with the following ideas for the Mantis graphical user interface:

- The interface should be easy for beginners to use (ideally without instructions), but should allow experienced users to work efficiently.
- A response should be given immediately for any user action, even if the response merely indicates that the user must wait.
- The interface should attempt to predict actions in order to reduce the user's workload, but should not be intrusive.
- The interface should cache information that is expensive to obtain so that redisplay is fast.
- Common interface controls (*e.g.*, the `emacs` text editing controls) should be supported—the choice of interface often depends on the expected experience of the user.
- The interface should present information concisely and allow the user to manage the screen area.
- Components of an interface should be easily identifiable as belonging to the interface, and any tasks that can be performed on more than one piece (such as dismissing a window) should be done in the same way.

- The interface should be visually pleasing.

This chapter explores each of these ideas, attempts to make them more concrete, and shows where they apply to the design of Mantis. At the end of the chapter, we discuss the use of the Tcl/Tk package to design and build user interfaces.

5.1 Simple, but Efficient

In our experience, a new user typically tries a new program as follows. First, the user starts the program to see what it can do. What it can do is often defined by what the user can get done in five or ten minutes without looking at any documentation on the program. If the program seems adequate by this measure, the user tries to perform a task using the new program. If at any point the interface seems restrictive or prevents the user from doing something important, the user gives up and goes back to an old program. If the new program manages to pass the test, the user may consider looking at the documentation for features that could speed up the work. The documentation is otherwise consulted only when the program has become a commonly used tool and the user runs into something for which the solution is not apparent after ten or twenty minutes of playing with the interface.

The point of this anecdote is that most users gauge tool functionality with the user interface, not the documentation. A good interface presents the user with a hierarchy of commands organized by usage. Common commands receive buttons or other control widgets, while less common commands appear in pull-down menus. For the advanced user, both command sets are accessible via keyboard shortcuts. Rarely-used commands are placed in a separate options window or on submenus. Using this hierarchy, an interface can introduce new users to the program yet still provide a valuable tool for the experienced user.

Mantis shifts the view slightly by splitting the hierarchy into sets of related tasks. The tasks are then grouped with distinct windows. Rarely used commands are placed into menus (e.g., changing between the source file, function, and disassembly selection methods is a menu operation). We are currently tracing the use of commands in the Mantis user interface to help structure the command hierarchy. We plan to use the trace data to develop a powerful set of keyboard shortcuts for the debugger.

5.2 Instantaneous Reaction

A good interface ensures that the user understands what the program is doing at all times; whether or not the interface recognized the last command should never be an open question.

When an action takes an extended period of time, the user should be made aware via some form of animation, either the cursor shape or a suitable icon. If the time can be quantified, the interface should provide an estimate of the remaining time or fraction of work until the task is complete.

In Mantis, many commands are passed on to the underlying `gdb` process, and we have no way of knowing how long that process will take to respond. The interface shows the user that the process is busy by writing a message describing the current action in the closest status area and oscillating the pendulum until the command finishes. Since only the `gdb` process is busy, the interface itself still interacts in a limited way with the user. Any command requiring the attention of the debugger results in the response that the interface is waiting for the previous command to complete, and the new command is stored for later execution. An arbitrary number of commands can be stacked in this fashion, but because of the methods of task management used in Mantis, stacks deeper than one are carried out in the reverse of the order given (a known bug).

To further aid the user, the cursor shape should reflect the function of the mouse buttons at the cursor location, and should be altered when the function changes. This kind of feedback helps provide users with hints as to the purpose of various areas without forcing them to read help files or manuals. If the task is one common to another program familiar to the user, the cursor shape from that program's interface should be used.

Mantis inherits cursor interface functionality automatically from Tcl/Tk. Buttons are highlighted when the cursor enters, and the cursor changes to an I-bar within entry boxes and text regions.

5.3 Intelligent Prediction

One measure of an interface is the time or amount of work required for the user to accomplish a task. When designing an interface to perform well under this metric, the designer must first understand the relative frequencies of tasks, then use those frequencies to make decisions about the placement and type of controls for each command. Once the designer has integrated the frequencies into the interface, further improvements can be made by trying to predict the user's next action. Intelligent prediction makes a user's work proceed more quickly, but the interface must not be overaggressive in its attempts to make predictions.

In Mantis, for example, the source file display switches to the current line when the processor halts or a new stack frame is selected. The time taken is negligible compared to the time required for the user to enter a command[1], and prediction saves the user the effort

[1]The time is noticeable in two cases. When a file is first read, a short delay occurs because `gdb` must

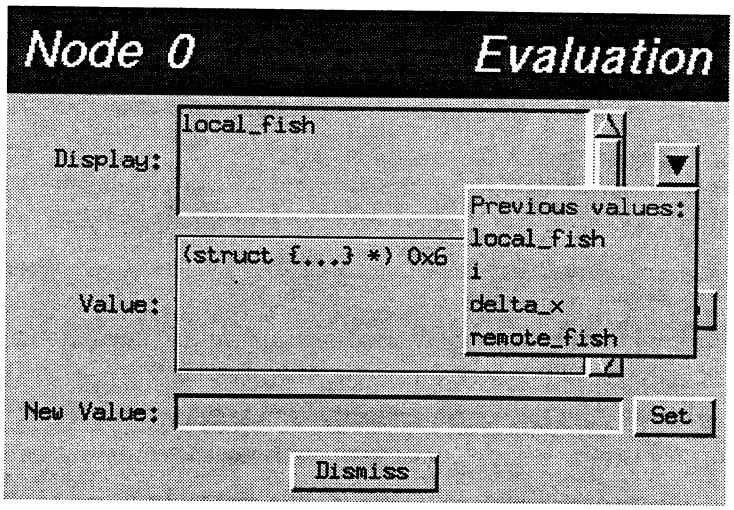


Figure 5.1: Evaluation window. A pull-down menu displays the cache of previously evaluated expressions.

of looking up the new function (requiring typing in most instances). On the other hand, the code previously displayed is accessible via the source menu.

In several cases, however, following a prediction is unwise. First, if the predicted task takes a long time (relative to the user's requesting the task to be performed), prediction is a bad idea because the user is forced to wait even when the task is undesired. Second, if the task destroys or replaces data that require a long time to recover, prediction is again bad. Finally, the interface should not follow a prediction if the user does not nearly always want the predicted task to be performed.

Evaluation of expressions, for example, is not always the right thing to do when a processor stops. Evaluation can be time-consuming on the CM-5 because of the Time Sharing Daemon and the lack of optimization in the modified gdb code. Forcing the user to wait for several evaluations each time a processor steps is unacceptable. Instead, Mantis allows the user to decide which, if any, expressions should be evaluated by providing the display window in addition to the normal evaluation window.

When the decision whether or not to act on predicted actions is not clear, it is best left up to the user, either via an option or through explicitly different means of accomplishing the task leading to the prediction.

load symbol information for the file. Also, when any function is disassembled during program execution, gdb reads the memory of the child process instead of reading the executable file, and memory reads can be quite slow under CMOST because of the Time Sharing Daemon bottleneck.

5.4 Caching Data

The general problem of caching information should not fall to the interface alone. If extensive calculations are required to generate a data view for the user, the program should also keep information for later reuse. Whether the latter type of caching is part of the interface or part of optimizing the program is debatable, but important. However, we are concerned here with reducing the time required for the user to generate information.

Entering a function name requires many keystrokes, for example. If the user is forced to retype the information many times, the interface becomes quite frustrating. Instead, the interface should maintain a cache of recently used values and let the user swap between them via either a menu or single keystrokes.

Mantis caches function names for function lookup and expressions for expression evaluation. The down-arrow menu button to the right of the entry box provides access to the cache, as shown in Figure 5.1. We currently maintain a separate data cache for each window, and information must be copied manually (via cut and paste) for use in another window. Alternatively, we might maintain a single cache of values for all windows.

Note that `gdb` also caches variable values and frame information while a processor is halted; all of this information becomes invalid when we allow the processor to continue.

5.5 Support for Known Interfaces

If a person encounters a familiar situation when trying out a new interface, the person naturally attempts to handle the situation with familiar means. Consider, for example, the process of text editing. Almost all users recognize one of the following interfaces:

- GNU Emacs
- Microsoft Word
- Word Perfect
- Borland Turbo Compilers (originally from WordStar)
- `vi`

Designing a new interface may seem like the right thing to do in some cases (after all, what could possibly be more natural than Alt-U to move the cursor up and Alt-D to move

it down?), but the designers are lucky if their intuition matches even half of the users' intuitions. Unless they plan to quickly replace every program with a version that supports the new interface or can convince large corporations to adopt their ideas, the new interface becomes yet another thing for the user to learn. It also becomes a stumbling block, since the user must remember to switch to the new interface when using the new program instead of simply remembering a single interface for a given task.

Since two common interfaces may assign different tasks to a particular key sequence, the designer should choose a single default interface for the new tool. The interface chosen should be determined by the user community—the interface familiar to the largest fraction of the users should be the default. Ideally, any interface already used by a significant fraction of the user community is available as an option, and the interface is also fully customizable. But since developing options and custom capabilities takes quite a bit of time, a first step is to simply allow commands from interfaces other than the default when they do not conflict with default commands. For example, if the default text editing interface makes no use of the arrow keys, the arrow keys should move the cursor in the appropriate directions.

For Mantis, we use the GNU `emacs` controls for text editing since GNU tools are common in the academic setting. In addition to the `emacs` controls, Mantis supports the common X-windows select and copy control (the left and middle mouse buttons, respectively) as well as typical PC cut and paste controls (**CTRL-X** and **CTRL-V**, respectively). The user can also move the cursor around with the arrow keys. Menus use the nearly universal left mouse button for normal operation, and the less common middle mouse button for tearoff (persistent) operation. Text selection and control buttons act with the left mouse button. For entry boxes, the **Tab** key moves the cursor to the next box and the **Return** key initiates an action related to the entry.

5.6 Efficient Information Display

The interface should generally allow the user to control the amount of space used by each window, and should scale and remove controls as necessary to fit within the desired space. When the program has more data than can be legibly displayed within the space allotted, the interface must provide mechanisms for the user to obtain the data that do not fit. Certain programs, including parallel debuggers and performance tools, have an amount of data proportional to the number of processors and must perform a significant amount of work to compress the data into a simple, but easily traversed, hierarchy.

Mantis uses the multiple window approach, creating a new window for each new set of data to be displayed. The advantage of this approach is that the data layout can be managed by the user by simply moving the windows about the screen and temporarily iconifying

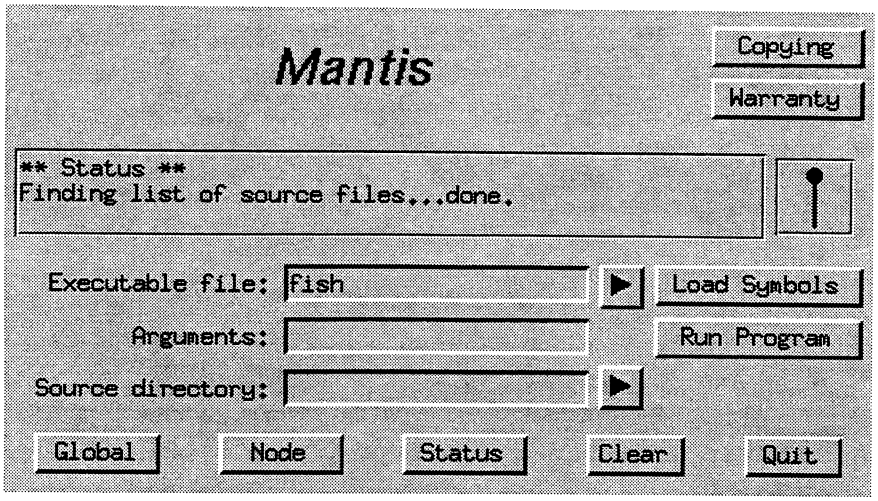


Figure 5.2: Main window. Control buttons found to the right of entry boxes are used for menus, file selection dialogs, and actions associated with the entry text.

windows as necessary. The disadvantage is that the windows use a large amount of screen space for labels, dismiss buttons, and blank space. We feel in retrospect that a better option is to add new fields to a window that displays data and to expand the window slightly when those fields are present. For example, one can imagine having evaluated expressions appear near the source file display in the node window instead of in a separate window.

The status window of Mantis provides an example of efficient data display. By representing the status of each node with color, and reducing the dimensions of the window to $\theta(\sqrt{P})$, Mantis allows the user to locate problems quickly and to focus on the problem with a single mouse click.

5.7 Common Design Pattern

The user should not have to read the fine print to recognize components of the same interface. Programs that use only one type of window can be distinguished easily by shape, size, and layout of the window. However, if multiple types of windows exist for an application, they should all have something in common. Further, the interface designer must select symbols and markings to be used for similar purposes across the entire interface.

The patterns used in designing an interface should be written down, both to help the designer to remember them and to later provide insight for the user who takes the time to read the documentation.

In Mantis, for example, all windows are labeled in a large, italicized font as shown in

Figure 5.2. Node subwindows such as the evaluation window in Figure 5.1 are then color-coded (pattern-coded on monochrome machines) to aid the user in finding related windows. A status region appears just below the label in each of the larger windows, with the inverted pendulum icon placed to the right. Spacing for borders and between widgets is the same in all Mantis windows, enhancing the similarities. Finally, all Mantis windows are dismissed using the button on the lower right hand corner. Dismissing the main window exits the program, of course.

A second example occurs with entry boxes: one or two control buttons are commonly placed to the right of each entry box, as with the three entries in the main window of Figure 5.2. An arrow symbol, pointing to the right for a file selection dialog or downward for a menu of recently cached values, sits closest to the entry box. A second button to the right is labeled with a brief textual description of the action performed by pressing **Return** in the entry.

Finally, the topological aspect of a widget frame is selected with the purpose of the widget in mind. Raised widgets indicate controls: buttons, menus, or scroll bars. Sunken widgets indicate text entry. Grooved and ridged widgets indicate data displayed for the user, with the ridge having the additional meaning that the user can then select data from within the widget to obtain further information. The matching of visual clues to purpose helps the user to understand intuitively how the interface works.

5.8 Pleasing to the Eye

Aesthetic value is the most qualitative aspect on our list and is therefore the most difficult to discuss. The interface should clearly allow users to make their own decisions about color schemes and patterns, but we must also consider a few finer points.

Color should not be abused. Color is effective for highlighting important information and helping to focus the user's attention, but using too many colors, or using bright colors too often, frustrates the user by making it hard to know where to look.

In general, a graphical interface should not be distracting. Frames and controls should be placed along straight lines unless they are intended to stand out. If several items appear on a line, spacing between the items should be even unless a natural subgrouping exists, in which case spacing for each level of the grouping hierarchy should be even.

5.9 Building Interfaces with Tcl/Tk

As we mentioned in Chapter 4, we wrote the Mantis user interface with the Tool Command Language (Tcl) [15][16] and Tcl toolkit (Tk) [17] developed by Ousterhout.

Tcl/Tk interfaces generally consist of two parts: a set of application-specific extensions to the language and a script to manage user interaction. On modern workstations, most operations run with adequate speed using only the interpreted script code. The remaining operations are written in C and included as new Tcl commands. Once the new commands are in place, they can be used in the scripts in the same way as any other Tcl command.

Because of its modular nature and the fact that the Tcl/Tk code was designed to be extensible, working with the system is very simple and straightforward. Full documentation is available with the package, and further explanation and examples of use can be found in [16].

For Mantis, we added one widget and five new commands. We created the source file display widget, which appears in Figure 5.3, by simply copying the code for the standard Tk listbox widget (one file) and changing the name, then making a few modifications. Figuring out the set of X calls needed to obtain the desired behavior was the most difficult part of the process. We then added the following commands to the Tcl interpreter:

a command to create a child process (the debugger process) with extra pipes for the process being debugged,

a command to check on the status of a child process in case the debugger process crashed,

a command to take asynchronous action when something appears on a pipe shared with the debugger or the process being debugged,

a command to animate the pendulum icon, and

a command to propagate breakpoint changes to all source file widgets.

The code for the source file widget is about twenty-five hundred lines, and the code for the commands comes to almost fifteen hundred. Although the C code is only broken up into two files, the functions separate easily into individual commands if desired. We have already reused commands written for Mantis several times, and we found the operation straightforward on each occasion.

The scripts for Mantis consist of sixteen files broken up by functionality and window type. Most files are a few hundred lines long, and the total length is over four thousand lines.

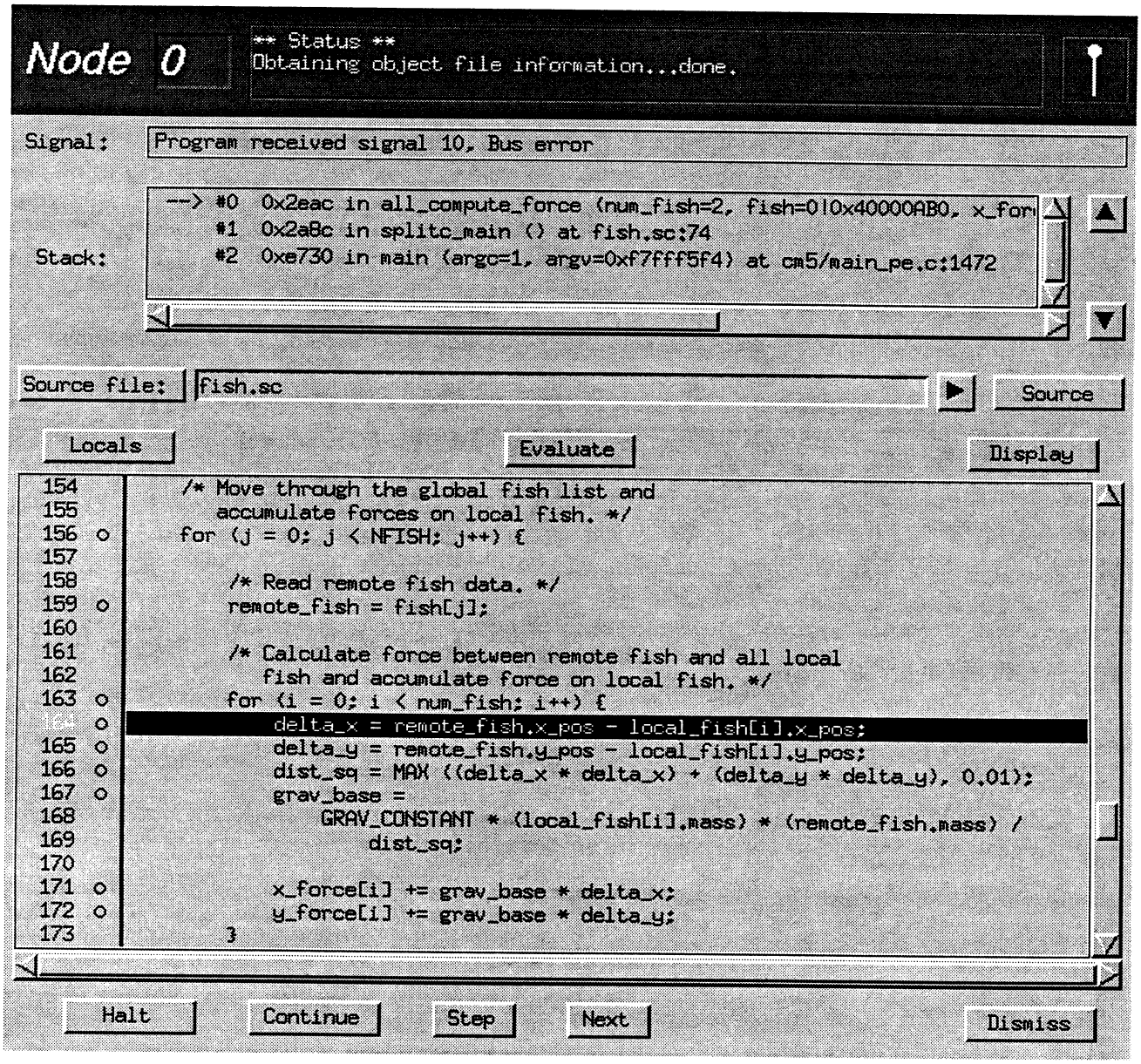


Figure 5.3: Node window. The source file display widget directs the user's attention to a problem.

Because of the script nature of the interface, we found it very easy to experiment and develop the interface to Mantis in real time. The time to make a change is dominated by the time it takes the programmer to type in the change instead of by compilation. The scripts also permit anyone who wants to customize the interface for their own purposes to do so. We doubt that the Mantis interface could be as clean as it is without the use of Tcl/Tk.

Chapter 6

Related Work

In this chapter, we explore other parallel debuggers, commenting on interface design, functionality, and portability, and examine other approaches to parallel debugging, noting the benefits and drawbacks of each approach.

6.1 Other Parallel Debuggers

This section discusses the merits of several parallel debuggers available to the general public. We begin with Panorama [12], created by John May to integrate the myriad techniques developed for parallel debugging in an easily portable and extensible manner. Next, we consider Node Prism [18], a product of Thinking Machines Corporation for the CM-5. Node Prism provides traditional debugging features and introduces a flexible method of control, emphasizing scalability for all operations. Finally, we examine TotalView, a debugger originally built by Bolt, Beranek, and Newman for the TC-2000 (the Butterfly) and now licensed by BBN for use on other platforms. TotalView emphasizes simplicity and consistency in its user interface, giving the user a very efficient debugging tool. All three present graphical interfaces to the user.

6.1.1 Panorama

Like Mantis, the Panorama debugger interface uses Tcl/Tk to enhance portability and extensibility. The Tcl/Tk interface communicates through a socket with the vendor-supplied debugger for the machine. Panorama provides a standard debugging interface across machines, using a "platform file" to translate between Panorama's commands and responses

and those used by the machine's debugger. Panorama currently provides platform files for the Intel Paragon and iPSC/860 and the Ncube/2.

Although Panorama allows for multiple windows and graphic visualizations of data and communication, the interaction is primarily textual. The user can create a separate window for each processor, and the commands allow for overriding the default processor (*e.g.*, "on all continue"), but the commands themselves must be typed. Source browsing, program control, and general expression evaluation are all performed using a command-line interface within a Tk text window. A more graphical interface could greatly reduce the time required to perform general tasks.

The visualization techniques, on the other hand, use primarily graphical mechanisms. Buttons on the main window create visualization windows, and the mouse manages alteration and expansion of information within the new windows. Windows in Panorama generally operate independently and only update displayed information by request. Included with Panorama are two windows to display message traffic, one statically and a second as a function of time. The user can add other visualizations easily by writing Tcl scripts, and the authors plan to extend the set of predefined visualizations as time permits.[1]

In addition to the traditional debugging capabilities, Panorama allows for tracing of communication activity for post-mortem debugging, supplying for each machine a library of tracing routines and preprocessor macros. Although the library is harder to port than the Tcl/Tk interface, the bulk of the code (over ninety percent) remains machine-independent. The value of the tracing facilities is questionable, however. Data presented in [12] indicate tracing overheads of between 5 and 65% of communication cost on the Intel iPSC/860 and the Ncube/2, depending primarily on message size and ignoring trace buffer overflow. Such large overheads can change the outcome of a race condition and prevent a bug from appearing. Further, and perhaps more importantly, real programs produce an enormous volume of trace data, creating I/O bottlenecks and storage difficulties. Tracing issues are discussed in more detail in Section 6.2.1.

The simple model used to build Panorama results in a highly portable platform for debugging, one readily extended to include new developments in visualization as they emerge. Panorama provides a standard interface with a minimum of effort.

Naturally, the model also has drawbacks. A significant amount of the debugging information in executable files is condensed or simply masked from the debugger user, and the results are then encased in English phrases and formatting characters. Though necessary for command-line debugging, the filtering process restricts the capabilities of programs like Panorama that must try to reverse the process in order to locate useful data. For example, information on source to executable mapping, communication library internals, scoping of

[1]The two visualizations discussed here were available in mid-1993, and others have been added since that time. In particular, a tool to display array values in color appears in the current version.

variables, and symbol lists are typically not directly available. Further, Panorama can only utilize the intersection of vendor debugger functionality, eliminating possibly useful tools, since otherwise Panorama could not fully support the standard interface on all platforms.

6.1.2 Node Prism

After developing Prism for debugging data parallel applications on the CM-5, Thinking Machines Corporation extended the capabilities of that debugger to support message-passing programs. Node Prism, described in [18], [19], and [20], gives the user traditional debugging capabilities combined with data visualization, performance tuning facilities, interesting methods of information compression, and a flexible method of control.

Node Prism, which currently runs only on the CM-5, uses capabilities of the CMOST operating system that were not available when we first conceived Mantis. In particular, Node Prism runs as a parallel process, monitoring each copy of the user's program with a separate debugger stub and retaining the one-to-one nature of the sequential debugger-to-program interface. The parallel nature of Prism makes scalability of commands fairly straightforward, since the bulk of the work can be done in parallel, leaving only small tasks for the host processor (which must still manage the user interface). A discussion of parallelizing the debugging tasks and deciding on the amount of information needed by the stubs can be found in [18].

In addition to the graphical user interface that we discuss here, Node Prism supports a command-line interface capable of managing data visualization and other graphics. Node Prism also integrates performance tuning tools with the debugging capabilities, and even provides a rudimentary interface to editors and compilers, but these features are beyond the scope of this document.

General interface

After Node Prism has read in the symbols for an executable and distributed the necessary information to the debugging stubs, the main window shown in Figure 6.1 becomes operational. The window divides into several regions: across the top is a menu bar, followed by a set of control buttons, a status region, a region for source file display, a scrollable feedback region, and a command entry line at the bottom. Prism automatically focuses the source display on the function `main`. The relative size of certain regions can be controlled by dragging one of the two small pins in the figure (one is just down and right from the `Collection` checkbutton, and the other is between the source and feedback regions. The sizes and other data are saved in an options file, making the user's customizations semi-permanent. The command entry line supports `emacs` controls, including moving forward

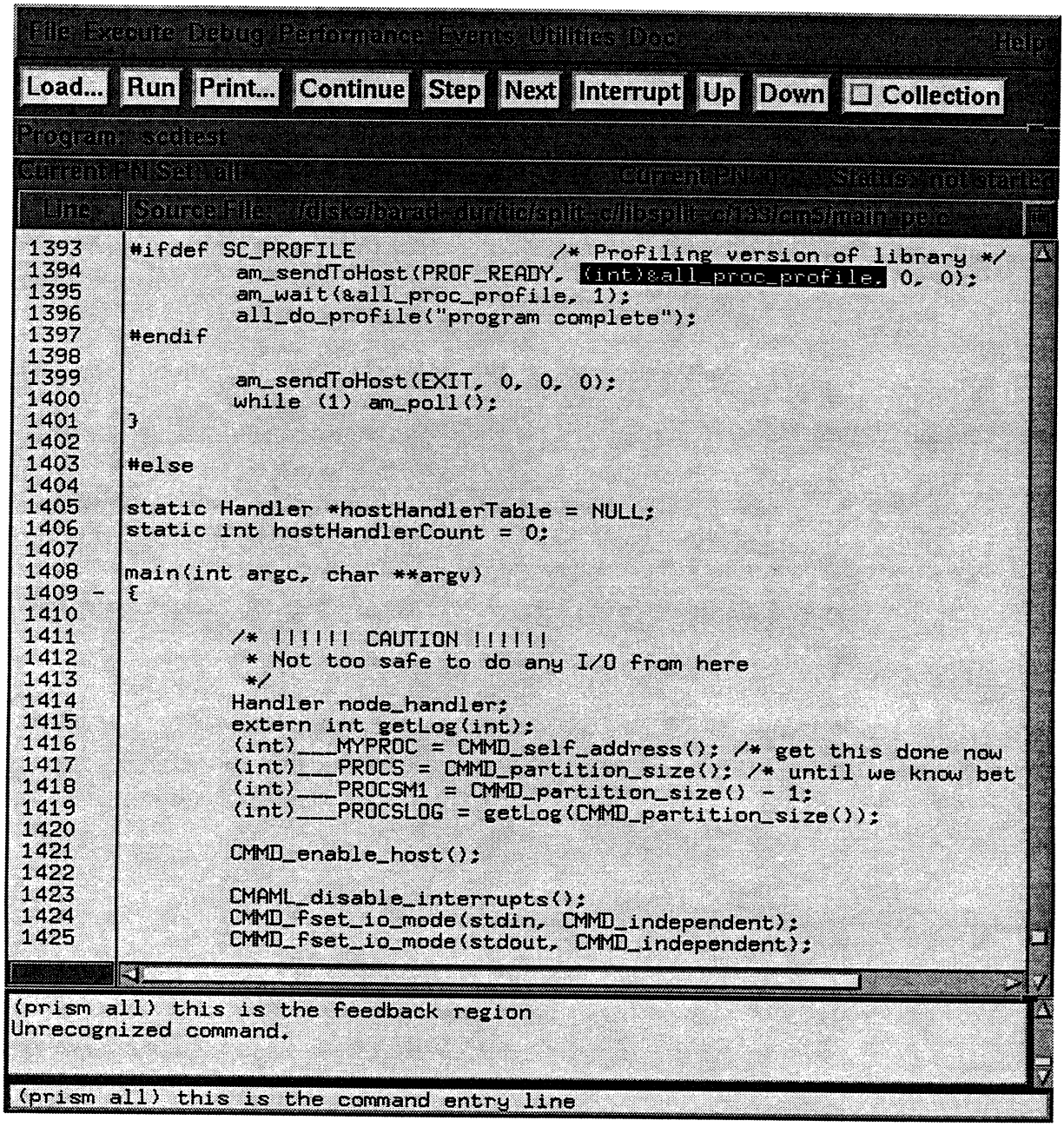


Figure 6.1: Main window of Node Prism. The source region automatically displays the start of `main`.

and backward through the command history with **CTRL-N** and **CTRL-P**. Unfortunately, the pleasantness of the general interface ends there.

The controls used to execute an action are often arcane, unnatural, or overcomplicated. Consider the keyboard shortcuts used for the row of buttons in Figure 6.1. Printable characters are sent to the command entry region, so the shortcuts cannot simply be letters. Prism chooses to use the function keys (`F1`, `F2`, etc.), a reasonable decision. The natural mechanism is to give the buttons and the shortcuts the same ordering and to use the unmodified function keys for the shortcuts. Prism, however, scrambles the order and requires the user to press the **CTRL** key for the shortcuts. The unmodified function keys are not used by Prism. By requiring the **CTRL** key and neglecting to make use of the natural keyboard to screen mapping of keys to buttons, Prism makes learning and use of the shortcuts more difficult than is necessary.

Once we manage to memorize the controls, we find that certain controls function sporadically or not at all. Deleting the character to the right of the cursor using **CTRL-D**, for example, ceases to work when the cursor is at the start of the line. Simultaneous two-button combinations on the mouse reportedly move through a list of previously displayed source files, but one direction of motion merely brings up a menu (the usual function of the second button pressed), while the other direction rarely works (we have seen it change files, but cannot get the control to work by choice). The list of files cannot be inspected, so the user must guess whether the lack of response indicates that the control does not work, the button presses were incorrectly performed, or the operation was executed on an empty list and resulted in no change.

As another example, we will try to select an expression in the source display region. Instead of allowing the user to point and click on expressions, Prism chooses to duplicate X-windows mechanisms exactly, using the left button click only to deselect text. Expression selection requires either pointing and dragging or double-clicking. In this instance, Prism has taken the idea of copying other interfaces too far, sacrificing simplicity for useless functionality.

Pointing to a variable and double-clicking, we must wait two seconds before Prism highlights the expression. During the two seconds, the interface gives no feedback, but freezes completely. If the user mistakenly assumes that Prism misread the double-click as two single clicks and double-clicks again, Prism will unhighlight the selection after freezing the interface for another three seconds and then, after yet another three seconds (for a total of eight), highlight the expression a second time. If the user is so bold as to click wildly, trying to force Prism to respond, the interface will merely crank away, making one change every two to three seconds, remembering every click received and never giving any indication of activity except for the periodic changes in highlighting.

We have no idea why Prism takes so long. The times given assume that Prism is the only

process running on the CM-5, prohibiting any explanation involving time sharing. When N other processes are sharing the machine, times are generally a factor of $N + 1$ longer. The debugged process, running or halted, counts toward the factor as well, so that if the user's program is running, the minimum selection response time is three or four seconds. Other interface activities such as scrolling the source suffer from the same delays in some instances, but react reasonably in others. The combination of huge delays and sporadic behavior can be incredibly frustrating.

Accepting the delays, we return to the selection process. Hopefully, the programmer has surrounded the expression with spaces, because Prism will include any commas, semicolons, or parentheses (and anything else except white space, it seems) along with alphanumeric strings when the user double-clicks. Since none of the Prism commands can filter out the extraneous bits of the string, the double-click method is often rendered useless.

At this point, most users return to the method of pointing and dragging to make selections. Hopefully, the user remembers to deselect expressions immediately, since otherwise Prism will force a two second delay while it unhighlights the old expression before allowing the user to see the bounds of the new selection.

Source browsing

Prism shows source code in the main window, which cannot be duplicated. The user can only display one source file at a time, perhaps a reasonable interface if the user can easily switch back and forth between pieces of code. Prism claims to keep a list of previously viewed files, but we were unable to use or view the list, as mentioned in the previous section. Even using other means of changing the displayed source, the three to five second delay required for Prism to show a new file prevents the user from quickly moving back and forth between pieces of code.

In order to change the file, the user must either type in the full filename at the command line or make use of a separate file selection window (shown in Figure 6.2) that lists the source files for the given executable. Prism can only display source files—the user must open a separate editor to view header files or other related information, even if the exact file name is known. Note also that if the user makes any mistake in the file selection window, the error information is routed to the feedback region of the main window (as is all error information). The response is clear when the user's attention is focused on the main window, but can go unnoticed when the user works with other windows. Also notice that in Figure 6.2, the file `bulk.h` appears many times. The problem is that the compiler inlines a function from `bulk.h` and must note the correct location of the source code in the debugging information. If the same function is used in several places, Prism lists each as an independent source file.

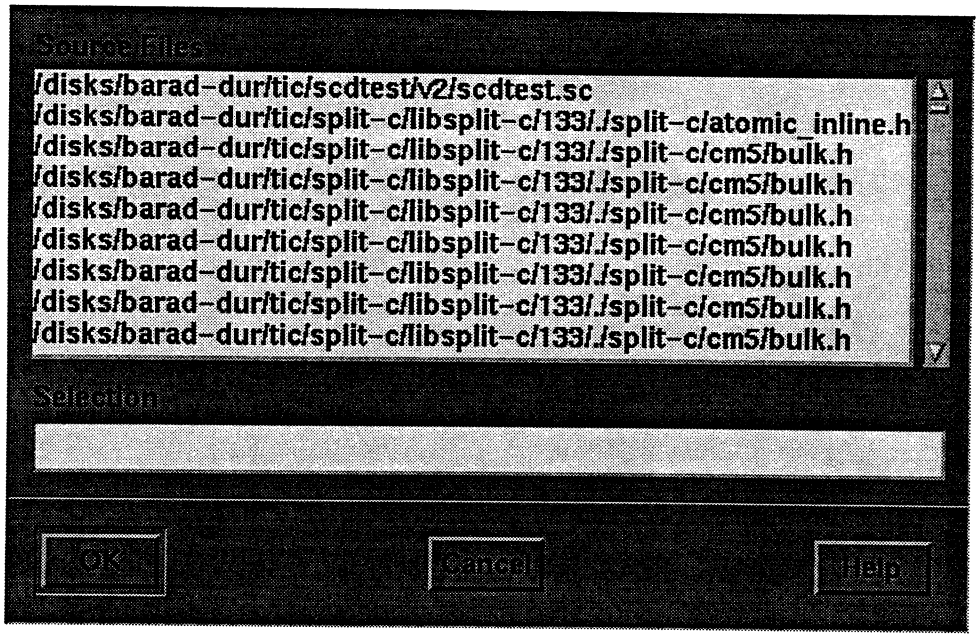


Figure 6.2: File selection window of Node Prism. Files with inlined code such as `bulk.h` appear once for each instance of inlining.

Finding a function is not much easier. The user can choose between the command-line, a separate function selection window, or a **SHIFT**-double-click mechanism that requires white space on both sides of the function name.

One redeeming feature of the source browsing interface is its ability to display assembly code simultaneously with the source code in two independent regions. Prism marks the assembly code with the line numbers of the original source file when available, and breakpoints can be set in either region.

Program control

Prism provides a very flexible scheme for program control, allowing the user to move easily between arbitrary subsets of processors. All operations involving program control, including breakpoints, interrupt, continue, and single-stepping the processors make use of the current processor subset. The syntax used to name the sets is drawn from data parallel array notation:

first : *last* : *step*

Prism extends the notation slightly to allow for arbitrary sets by linking multiple array specifiers together with commas. The processor set

1:5:2, 33:36

would include processors 1, 3, 5, and 33 through 36, for example.

A separate window allows the user to change between defined sets or to define new ones. Although some of the visualization tools allow the user to create a new set without typing, we could not find a graphical means of defining an arbitrary set. On the other hand, the command line and processor set creation window provide powerful textual tools for set definition, including set operations (union, intersection) and the ability to define a set based on the result of a Boolean expression evaluated on the processors.

Prism also handles breakpoints well. Breakpoints can be created and destroyed using a single click near a line number to the left of the source region. The cursor becomes a "B" when it enters the region, and the same symbol is used to mark existing breakpoints. No information is given as to where breakpoints may be set, however, so the user must watch the feedback region if clicking results in no response. More detailed breakpoint creation is available via a separate window. The user can give a condition and count to check before stopping and can provide an arbitrary set of actions to execute when the breakpoint is reached.

Data display and visualization

Data display in Prism uses a single interface with several options. Prism automatically collects the results from each processor in the processor set and allows the user to choose between display in the feedback region or in a dedicated window.

Prism compresses any text entering the feedback region into a list of unique responses. Each response is tagged with a subset of the processor set to tell the user which processors gave a particular response. Changes in processor status and other feedback information also use the compression process, requiring less effort for the user to understand possibly important information.

Dedicated display windows can be marked as "snapshots," in which case Prism preserves the data from modification by the program. Non-snapshot windows are partially obscured with diagonal slashes when the data becomes stale, but the user can request that the data be updated from a menu in the display window.

A second menu in the display window allows the user to gather statistics and to change the representation of the data, selecting from one of several interesting options. Besides the

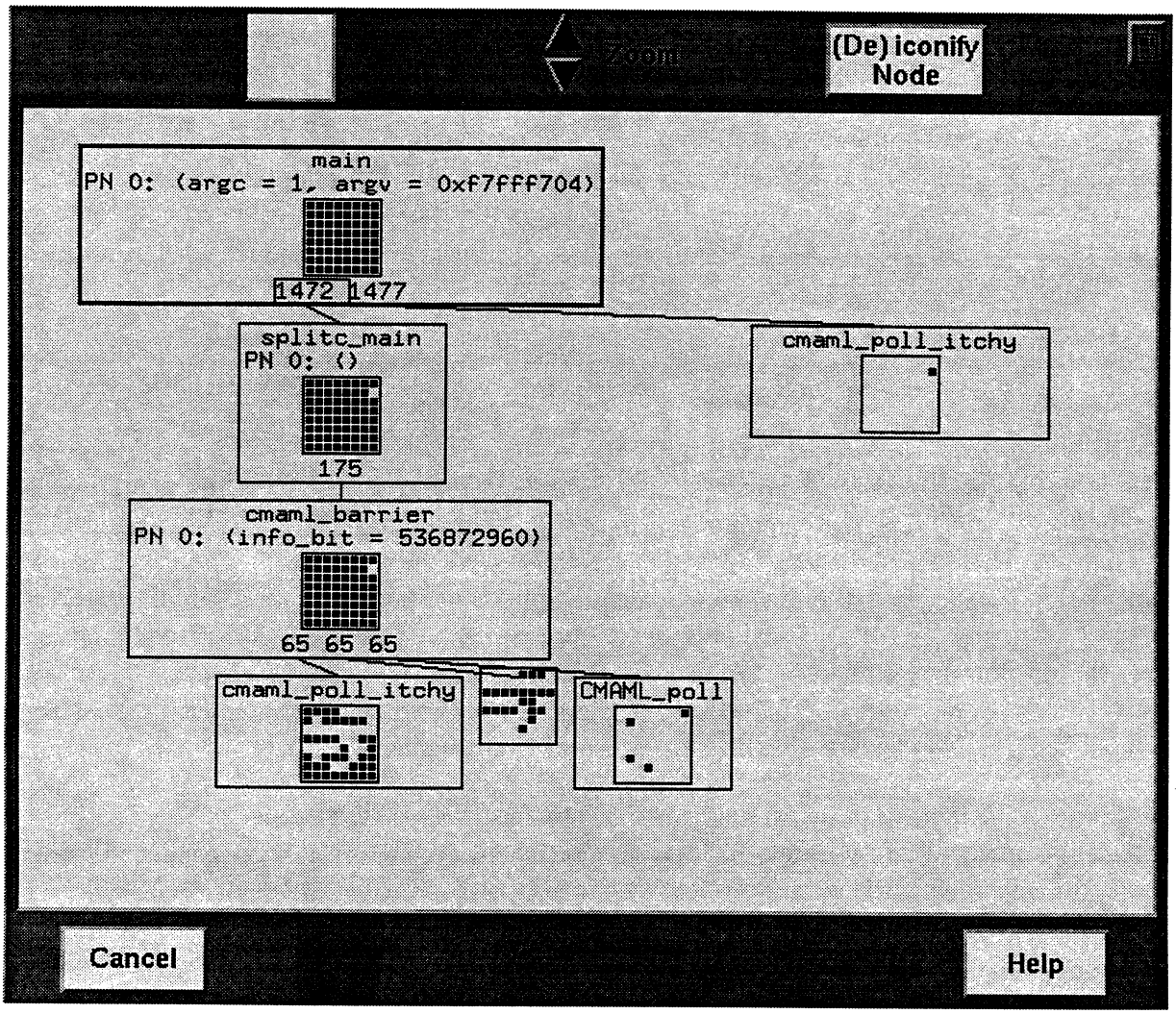


Figure 6.3: Stack trace comparison window of Node Prism. The second bug from Chapter 2 causes the first branching.

standard textual representation, Prism supports several graphical displays. For example, a color map display uses a heat color scheme (blue is cold or low-valued, red is hot or high-valued) to represent each data point as a block of color, allowing the user to quickly scan the data for appropriate patterns.

Prism also supports a graphical views of structures, representing each structure as a separate box. Pointers within the structure are represented as small dots—the user can click on the dot and expand the pointer into a new value or structure box. The interface displays dynamic arrays, but the programmer must add a function and recompile to set the size of a dynamic array.

One particularly useful tool is a window displaying a tree of stack traces, as shown in Figure 6.3. In the figure, we have taken the second bug from Chapter 2 and displayed the stack trace tree after halting all processors. Nodes and subtrees in the window can be iconified to better manage the display space, and the amount of data shown in each node can be controlled using the zoom buttons. The figure shows the highest zoom level, giving not only the arguments to each call but a map of processors which entered that call. By double-clicking on a node, the user can change the current processor set to the set of processors in the node selected.

Summary

Prism is a highly functional debugger that is already integrated with many useful tools. By placing debugging stubs on each node, Prism maintains the one-to-one relationship between the debugger and the program threads and retains the ability to support asynchronous notification and useful abstractions such as conditional breakpoints. Processor set specification enhances the power of traditional debugging operations, and compression of feedback reduces the amount of data scanned by the user. A selection of visualization tools are available along with a uniform data display interface. A graphical tool for comparing stack traces rapidly focuses the user's attention on problem areas of the program.

The main difficulty in using Prism is the interface. Whether because of the TS Daemon's scheduling methods or because of bugs in the interface, Prism at times runs inordinately slowly. Combined with overcomplicated controls and complete lack of feedback in some cases, the speed problems result in an interface that can be almost painful to use.

Prism is also only available on the CM-5, meaning that users must choose another debugging tool if they wish to port the code to other parallel platforms. We see no reason, however, for TMC not to follow BBN's example with TotalView and license Node Prism for use on other machines.

6.1.3 TotalView

The TotalView debugger from BBN presents the user with a consistent interface to all functionality. The user can open a separate source display window for each process in the parallel program and perform various data visualization tasks.

TotalView assigns each of the three mouse buttons a general purpose, and the buttons act appropriately in all situations. The left button makes selections from lists and text, the middle button drives the menu system, and the right button "dives" deeper into information. In the source region, for example, the left button can be used to select expressions from the

source, the middle button brings up a menu for data display and other options, and the right button evaluates the expression or brings up the function selected.

Despite the beneficial features of the TotalView interface, learning to use the debugger does take some effort. Many operations require keyboard shortcuts or menus, offering no means to gradually introduce users with more visually-oriented means such as buttons and entry boxes. Although consistency aids a new user in the learning process, the interface caters primarily to the expert user who prefers to use the keyboard over the mouse.

TotalView had a strong influence on the development of Mantis. The popularity of TotalView over other debuggers convinced us of the key role played by the user interface in making a tool productive or useless.

6.2 Different Approaches to Debugging

This section discusses a few other approaches to debugging, including tracing of communication for deterministic replay, animation of program activity, and dynamic instrumentation techniques.

6.2.1 Tracing communication

A large fraction of the parallel debugging community believes that tracing and deterministic replay will prove essential to debugging parallel programs. Another large fraction, including the author, recognizes the potential usefulness but maintains that the perturbations to the program and the sheer volume of trace data generated by real programs will continue to make tracing an impractical alternative.

Tracing generally consists of adding small sections of code to each interprocessor communication call at the library level. The code records interesting information such as the time according to the local processor and the type, size, and destination of the message. The tracing code writes the data into a special buffer and flushes the buffer to disk when full.

Performance tuning tools also use tracing. Paragraph [9][10], for example, displays animated graphical views of trace files generated using the Portable Instrumented Communication Library (PICL). PICL supports optional tracing of communication events as defined above. The PICL library consists of two portable layers on top of a third, non-portable layer that makes use of the machine-dependent message-passing primitives. The authors of ParaGraph give a favorable view of the overhead and perturbation effects of PICL, but some users are less sure [8].

For debugging purposes, the most interesting and perhaps useful view in Paragraph, called the "Task Gantt Graph," shows the tasks performed by each processor, the y-axis, as a function of time, the x-axis. Tasks are defined by the user: for each section of code comprising a task, the user selects a numerical ID and inserts special PICL calls marking the start and end of the section. The ID's need not be unique, but sections with the same ID appear in the same color. Paragraph assigns a different color to each task up to the sixth[2] and recycles colors for later tasks. The graph is equivalent to a sequential profiler that retains time data for each processor instead of collapsing the data into averages. When debugging, the graph could be used to illustrate the bulk synchronous block structure of the program, highlighting violations of the blocks for the user. The graphical representation helps call attention to anomalies. Unfortunately, Paragraph requires that the user recompile the program to change the definition of tasks.

Trace data for the Gantt graph is minimal, but in order to provide a deterministic replay of the program, we must collect much more data. In fact, we must record data for all interprocessor communication. Selecting the size of the trace buffer for full traces can be tricky. The smaller the buffer, the more frequently the program suffers an extremely slow disk access as the buffer flushes. The larger the buffer, the less memory can be used by the parallel program. In our experience with tracing Split-C communication, real programs[3] often generate more data than even the largest buffers can hold (more than 10MB per processor in a sample human genome problem), causing repeated flushes via slow I/O channels and altering the temporal behavior of the program.

Even for the class of programs that perform sufficiently little communication that tracing might be feasible, the amount of time taken by the tracing code is generally comparable to the faster message-passing calls. As machines progress, the limiting factor for both tracing and communication will become the cost of accessing the lowest level of the memory hierarchy. The trace must read the clock and write the trace data, both of which miss higher levels of the hierarchy, while communication calls must send a short message across the memory bus to a DMA controller or full network interface processor. The time required on future machines will thus continue to be comparable, and tracing will continue to disrupt program behavior.

6.2.2 Animation

Several parallel debugging projects have suggested the use of animation as a tool to help the user understand the program's interactions. The general approach requires that the user spend time to augment the program with appropriate icons and annotations to allow the

[2]Paragraph supports sixty-four colors in later versions, but distinguishing sixty-four colors is hard.

[3]By "real programs," we mean programs written by people other than the language designers to solve problems that they want to solve (*i.e.*, not benchmarks, kernels, or example programs).

animation package to display the results. The approach also entails tracing, often beyond that required for deterministic replay.

We doubt that parallel programmers will frequently use animation. Only large programs merit the additional overhead of writing and debugging animation extensions, but large programs generate an enormous volume of trace data, invariably changing the behavior of the program.

6.2.3 Dynamic instrumentation

The process of dynamic instrumentation involves the addition of short instruction sequences to an existing executable. Parallel performance tuning tools like Paradyn [13] avoid perturbing the program except during short intervals by dynamically inserting and removing performance instrumentation.

Dynamic instrumentation also appears in some debuggers to supplement debugging capabilities by compiling extensions to the program immediately, allowing the user to make small modifications to the program without a full recompilation process. The modifications generally include minor changes to source code and creation of fast conditional breakpoints and trace points.

Chapter 7

Future Work

This chapter briefly outlines the future of Mantis, discussing plans and possibilities for extension. In Section 7.1, we consider additions to the user interface. Section 7.2 discusses the benefits of adding library support for debugging. Section 7.3 describes a way to use Mantis as a starting point for other projects. Finally, Section 7.4 details our current efforts in porting Mantis to other platforms.

7.1 Extensions to the Interface

The first addition to Mantis will involve some form of data visualization. Most parallel debuggers provide support for graphical representation of data to help the user to recognize incorrect results and to locate the source of the error. Simple methods of adding support for visualization are to create a file or socket interface to a visualization package like AVS or to instrument the code to work with a package like VIGL (described in [6]). Alternatively, we might build visualizations directly into Mantis using Tcl/Tk scripts as Panorama[12] does, trading faster functionality for better extensibility.

A related extension involves displaying structures and dynamic arrays as objects and allowing the user to expand and contract links. Although many debuggers already support this type of display, several features seem to be lacking from all of them. For example, coercing objects into related templates (other structures) should be straightforward, perhaps even allowing the user to alter templates and define new ones. Often related structures can be differentiated by means of a field near the top of the object, and we might allow the user to define the relationship within the debugger. Mantis should also accept an expression for the length of dynamic arrays. Some users may prefer to build extra code into their programs to support our new displays, but others will prefer to work through the debugger. For the

latter group, Mantis should save display information for later use, learning about the user's preferences and shortening the time-consuming process of configuration.

Another useful feature would allow the user to save and restore information on the state of Mantis, including window layout and breakpoint data. A programmer working on a single project might be saved considerable effort if Mantis could automatically manage windows and duplicate the breakpoints of previous debugging sessions.

Finally, we would like Mantis to distinguish between user code and Split-C library code when we examine a processor. Special functions could be added to give the user a meaningful indication of events when the processor operates inside the Split-C library. For example, if the processor stops in a message handler, Mantis could provide details on the message type, source, and contents. Mantis could also differentiate between processors busy doing computation and those simply waiting for data from other processors at any point in time. More interesting results can be achieved by adding support within the library, as we discuss in the next section.

7.2 Library Support

Another direction for future work involves modifying the Split-C library to support new debugging operations. Until now, we have avoided changes to the library for two reasons: they require that the user recompile the program (or at least relink it), and the more interesting changes inevitably perturb the program. As we discussed in Section 6.2.1, even minor perturbations can cause changes in timing and make bugs disappear, leaving the user frustrated. However, when used optionally and with the understanding that they may not work, the methods that follow make the user's task easier.

First we might add a piece of code to handle incoming messages, simply a loop that constantly polls the network. By calling the loop on a halted processor, Mantis could allow the user to stop one processor without interfering with the computations of others. The user must exercise caution in using the option to avoid corrupting data on a processor halted in a critical section.

Recording the location of the last barrier passed might help the user locate bugs with the bulk synchronous block strategy. At the cost of a minor perturbation, Mantis could present a compressed representation of the various barriers, allowing the user to rapidly detect any inconsistency. We might couple this extension with a global option to step to the next barrier, enhancing the capabilities of the bulk synchronous view.

We could also add partial message tracing to the library. Full tracing and storage is not feasible with real programs, and any software tracing results in significant perturbation of

the program—the time to fill a trace record is comparable to the time to send a message on the CM-5. Nevertheless, in some cases we can gain valuable insight by examining the last few messages sent and received by each processor. Mantis can recognize library functions in the message data and present the messages using Split-C terminology: "Processor 17 put (int)0x125411 into my_array[102]," for example. Finding an appropriate name for the storage location might be difficult, however.

7.3 Mantis as a Subprogram

Split-C provides the direct control and simple source to executable translation required for programmers to fully optimize parallel programs, but it can also serve as a target language. Higher-level parallel languages and libraries can compile into Split-C, avoiding the cost of developing separate translations for each architecture.

Figure 7.1 shows how a high-level language and debugger might interact with the Split-C compiler and Mantis. The user in the upper right portion of the figure writes source code in the high-level language and then compiles the code using the high-level compiler. The compiler produces not only source code in Split-C, but a mapping from the original source to the Split-C source. The mapping is analogous to the debugging information included in the executable file, but appears instead in a separate file. The Split-C compiler is then invoked to create an executable.

For debugging, the high-level debugger uses Mantis and the mapping produced by the high-level compiler. Commands that refer to high-level constructs are translated into Split-C via the mapping before being sent to Mantis. Similarly, Mantis' responses undergo the inverse translation from Split-C into the high-level language abstractions before returning to the user.

The Castle project includes several high-level languages and library abstractions; hopefully they will be able to use Mantis in the fashion outlined here to simplify the process of building debuggers.

7.4 Porting to Other Platforms

Work has already begun on a version of Mantis for networks of workstations with the NOW project at Berkeley. We have upgraded the interface significantly, adding X configuration options and other features and cleaning up the code. We have created a sequential version of Mantis in preparation for the new platform, which will utilize the debugging model described

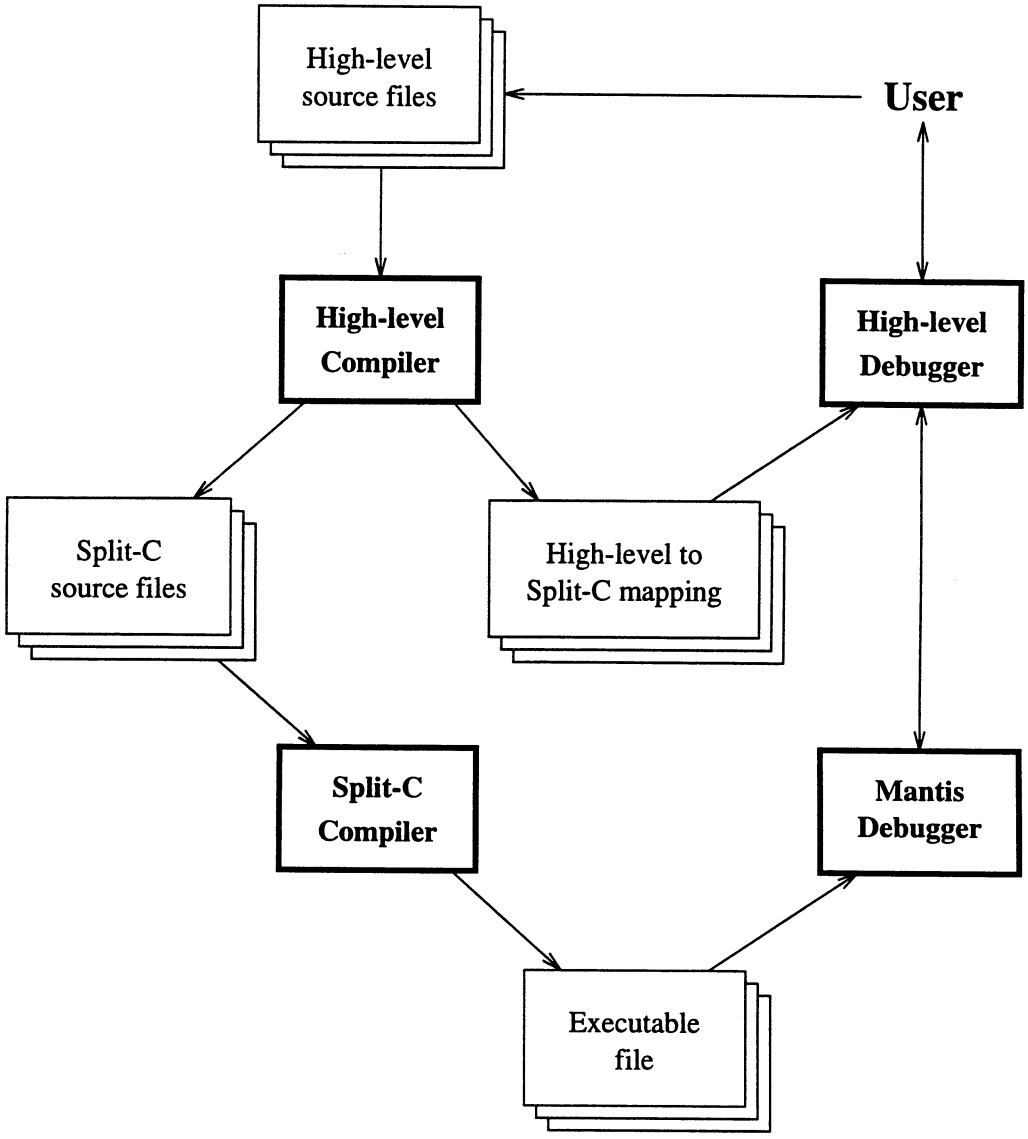


Figure 7.1: High-level language built using Split-C and Mantis. The user need only interact with the high-level compiler and debugger.

in Section 4.2.1. The updated graphical user interface runs both with the sequential version and with the CM-5 version of the underlying `gdb` debugger, allowing the user to have the same debugging interface for sequential and parallel debugging needs.

The new model will solve the problem of asynchronous notification and all of the difficulties arising from that problem, as outlined in Sections 3.8 and 4.2.4. Further, the new version will match the requirements of more typical systems, and should ease the process of porting Mantis in the future.

Appendix A

Code for Fish and Gravity

```
1   #include <split-c/split-c.h>
2   #include <split-c/control.h>
3   #include <split-c/com.h>
4   #include <math.h>
5   #include <malloc.h>
6
7
8   #define NFISH              100      /* number of fish */
9   #define T_FINAL           10.0      /* simulation end time */
10  #define GRAV_CONSTANT      1.0      /* proportionality constant of
11                                        gravitational interaction */
12
13  /*
14     This structure holds information for a single fish, including
15     position, velocity, and mass.
16  */
17
18  typedef struct {
19      double x_pos, y_pos;
20      double x_vel, y_vel;
21      double mass;
22  } fish_t;
23
24
25  /* These procedures perform subtasks for splitc_main. */
26  void all_init_fish (int num_fish, fish_t *local_fish);
27  void all_compute_force (int num_fish, fish_t *spread fish,
28                          double *x_force, double *y_force);
29  void all_move_fish (int num_fish, fish_t *local_fish, double delta_t,
30                      double *x_force, double *y_force,
31                      double *max_acc_ptr, double *max_speed_ptr,
32                      double *sum_speed_sq_ptr);
```

```
33
34
35      /*
36         Simulate the movement of NFISH fish under gravitational attraction.
37         splitc_main initializes the fish, then enters the main loop.  The loop
38         divides into three synchronous blocks for force computation, fish
39         motion, and data collection.
40      */
41
42      splitc_main ()
43      {
44          double t = 0.0, delta_t = 0.01;
45          double max_acc, max_speed, sum_speed_sq, mnsqvel;
46          fish_t *spread fish, *local_fish;
47          int num_fish;
48          double *x_force, *y_force;
49
50          /* Allocate a global spread array for the fish data set and
51             obtain a pointer to the local portion of the array.  Then
52             find the number of fish owned by this processor. */
53          fish       = all_spread_malloc (NFISH, sizeof (fish_t));
54          local_fish = tolocal (fish);
55          num_fish   = my_elements(NFISH);
56
57          /* Allocate force accumulation arrays. */
58          x_force = (double *)malloc (num_fish * sizeof (double));
59          y_force = (double *)malloc (num_fish * sizeof (double));
60
61          /* Initialize the fish structures, then synchronize
62             to ensure completion. */
63          all_init_fish (num_fish, local_fish);
64          barrier ();
65
66          /* Enter the main loop. */
67          while (t < T_FINAL) {
68
69              /* Update time. */
70              t += delta_t;
71
72              /* Compute forces on fish due to other fish, then synchronize to
73                 ensure that no fish are moved before calculations finish. */
74              all_compute_force (num_fish, fish, x_force, y_force);
75              barrier ();
76
77              /* Move fish according to forces and compute rms velocity.
78                 Note that this function is completely local. */
79              all_move_fish (num_fish, local_fish, delta_t, x_force, y_force,
80                             &max_acc, &max_speed, &sum_speed_sq);
81
82      /*
83         BUG: The programmer used all_reduce_to_one_d* instead of
```

```
84          all_reduce_to_all_d*, resulting in different time steps on each
85          processor that eventually caused synchronization errors and
86          hung the program.
87      */
88
89              /* Compute maximums across all processors and sum for rms speed
90                 (this will synchronize the processors). */
91              max_acc      = all_reduce_to_one_dmax (max_acc);
92              max_speed    = all_reduce_to_one_dmax (max_speed);
93              sum_speed_sq = all_reduce_to_one_dadd (sum_speed_sq);
94              mnsqvel      = sqrt (sum_speed_sq / NFISH);
95
96              /* Adjust delta_t based on maximum speed and acceleration--this
97                 simple rule tries to insure that no velocity will change
98                 by more than 10%. */
99              delta_t = 0.1 * max_speed / max_acc;
100
101             /* Print out time and rms velocity for this step. */
102             on_one {printf ("%15.6lf %15.6lf;\n", t, mnsqvel);}
103         }
104     }
```

```
105
106
107     /*
108         Place fish in their initial positions.
109     */
110
111     void
112     all_init_fish (int num_fish, fish_t *local_fish)
113     {
114         int i, n;
115         double total_fish = PROCS * num_fish;
116
117         for (i = 0, n = MYPROC * num_fish; i < num_fish; i++, n++) {
118             local_fish[i].x_pos = (n * 2.0) / total_fish - 1.0;
119             local_fish[i].y_pos = 0.0;
120             local_fish[i].x_vel = 0.0;
121             local_fish[i].y_vel = local_fish[i].x_pos;
122             local_fish[i].mass  = 1.0 + n / total_fish;
123         }
124     }
125
126
```

```
127     /*
128        Compute the force on all local fish according to the 2-dimensional
129        gravity rule,
130            F = d * (GMm/d^2),
131        and add it to the force vector (fx, fy).  Note that both fish and
132        both components of the force vector are local.
133     */
134
135     void
136     all_compute_force (int num_fish, fish_t *spread fish,
137                        double *x_force, double *y_force)
138     {
139         int i, j;
140         fish_t *local_fish, remote_fish;
141         double delta_x, delta_y, dist_sq, grav_base;
142
143         /* Clear forces on local fish. */
144         for (i = 0; i < num_fish; i++) {
145             x_force[i] = 0.0;
146             y_force[i] = 0.0;
147         }
148
149     /*
150        BUG: The programmer forgot to initialize local_fish before
151        using it below.
152     */
153
154         /* Move through the global fish list and
155            accumulate forces on local fish. */
156         for (j = 0; j < NFISH; j++) {
157
158             /* Read remote fish data. */
159             remote_fish = fish[j];
160
161             /* Calculate force between remote fish and all local
162                fish and accumulate force on local fish. */
163             for (i = 0; i < num_fish; i++) {
164                 delta_x = remote_fish.x_pos - local_fish[i].x_pos;
165                 delta_y = remote_fish.y_pos - local_fish[i].y_pos;
166                 dist_sq = MAX ((delta_x * delta_x) + (delta_y * delta_y), 0.01);
167                 grav_base =
168                     GRAV_CONSTANT * (local_fish[i].mass) * (remote_fish.mass) /
169                         dist_sq;
170
171                 x_force[i] += grav_base * delta_x;
172                 y_force[i] += grav_base * delta_y;
173             }
174         }
175     }
176
```

```
177
178     /*
179         Move fish one time step, updating positions, velocity, and
180         acceleration.  Return local computations of maximum acceleration,
181         maximum speed, and sum of speeds squared.
182     */
183
184     void
185     all_move_fish (int num_fish, fish_t *local_fish, double delta_t,
186                    double *x_force, double *y_force,
187                    double *max_acc_ptr, double *max_speed_ptr,
188                    double *sum_speed_sq_ptr)
189     {
190         int i;
191         double x_acc, y_acc, acc, speed, speed_sq;
192         double max_acc = 0.0, max_speed = 0.0, sum_speed_sq  = 0.0;
193
194         /* Move fish one at a time and calculate statistics. */
195         for (i = 0; i < num_fish; i++) {
196
197             /* Update fish position, calculate acceleration, and update
198                velocity. */
199             local_fish[i].x_pos += (local_fish[i].x_vel) * delta_t;
200             local_fish[i].y_pos += (local_fish[i].y_vel) * delta_t;
201             x_acc = x_force[i] / local_fish[i].mass;
202             y_acc = y_force[i] / local_fish[i].mass;
203             local_fish[i].x_vel += x_acc * delta_t;
204             local_fish[i].y_vel += y_acc * delta_t;
205
206             /* Accumulate local max speed, accel and contribution to
207                mean square velocity. */
208             acc = sqrt (x_acc * x_acc + y_acc * y_acc);
209             max_acc = MAX (max_acc, acc);
210             speed_sq = (local_fish[i].x_vel) * (local_fish[i].x_vel) +
211                 (local_fish[i].y_vel) * (local_fish[i].y_vel);
212             sum_speed_sq += speed_sq;
213             speed = sqrt (speed_sq);
214             max_speed = MAX (max_speed, speed);
215         }
216
217         /* Return local computation results. */
218         *max_acc_ptr      = max_acc;
219         *max_speed_ptr    = max_speed;
220         *sum_speed_sq_ptr = sum_speed_sq;
221     }
```

Bibliography

[1] T. Anderson, D. Culler, J. Demmel, J. Feldman, S. Graham, P. Hilfinger, A. Sangiovanni-Vincentelli, K. Yelick, "The Castle Project: A Proposal to the Department of Energy for Integrated Parallel Scientific Computing"

[2] H. E. Bal, A. S. Tanenbaum, M. F. Kaashoek, "Orca: a language for distributed programming," *SIGPLAN Notices*, Vol. 25, No. 5, pp. 17-24, May 1990.

[3] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick, "Parallel Programming in Split-C," *Proceedings of Supercomputing '93*, pp. 262-73, Portland, Oregon, November 1993.

[4] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, J. Wawrzynek, "Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine," *SIGPLAN Notices*, pp. 164-75, April 1991.

[5] K. Dewdney, "Computer Recreations: Sharks and fish wage an ecological war on the toroidal planet Wa-Tor," *Scientific American*, December 1984.

[6] A. B. Downey, "VIGL 2.3: Visualization Graphics Library," U. C. Berkeley Technical Report#CSD-93-764, November 1994.

[7] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, *Solving Problems on Concurrent Processors*, Vol. I, Ch. 17, pp. 307-25, Prentice Hall, Englewood Cliffs, New Jersey.

[8] J. M. Francioni, D. T. Rover, "Visual-Aural Representations of Performance for a Scalable Application Program," *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*, pp. 433-40, Williamsburg, VA, April 1992.

[9] M. T. Heath, "Recent developments and case studies in performance visualization using ParaGraph," *Proceedings of Performance Measurement and Visualization of Parallel Systems*, pp. 175-200, Moravany, Czechoslovakia, October 1992.

[10] M. T. Heath, J. A. Etheridge, "Visualizing the performance of parallel programs," *IEEE Software*, Vol. 8, No. 5, pp. 29-39, September 1991.

[11] S. E. Lucco, D. P. Anderson, "Tarmac: a language system substrate based on mobile memory," *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 46-51, 1990.

[12] J. May, F. Berman, "Panorama: a portable, extensible parallel debugger," *SIGPLAN Notices*, pp. 96-106, December 1993.

[13] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, T. Newhall, "The Paradyn Parallel Performance Measurement Tools," University of Wisconsin at Madison Technical Report, available from `http://www.cs.wisc.edu/ paradyn/papers.html`.

[14] R. B. Osborne, "Speculative computation in Multilisp: an overview," *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pp. 198-208, 1990.

[15] J. K. Ousterhout, "Tcl: An Embeddable Command Language," *Proceedings of the USENIX Winter Conference*, pp. 133-46, 1990.

[16] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[17] J. K. Ousterhout, "An X11 Toolkit Based on the Tcl Language," *Proceedings of the USENIX Winter Conference*, pp. 105-15, 1991.

[18] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, R. Title, "A scalable debugger for massively parallel message-passing programs," *IEEE Parallel & Distributed Technology: Systems & Applications*, Vol. 2 No. 2, pp. 50-6, Summer 1994.

[19] Thinking Machines Corporation, *Prism User's Guide*, Version 1.2, March 1993.

[20] Thinking Machines Corporation, *Prism 2.0 Release Notes*, May 1994.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," *Proceedings of the International Symposium on Computer Architecture*, 1992