The Mantis Parallel Debugger

Steven S. Lumetta and David E. Culler *

Abstract

Parallel tools often fail to integrate effectively with other parallel systems, exacerbating the inherent difficulty of programming a massively parallel machine. Successful parallel debugging requires the rethinking of traditional debugging goals in the context of parallelism. The following four objectives: rapid focus, scalability, economy of presentation, and portability, represent the more difficult aspects of parallel debugging.

In light of these specific goals, this paper presents Mantis, a graphical debugger for parallel programs. Mantis targets the broad class of parallel programs known as bulk synchronous SPMD programs and provides support for Split-C, a parallel extension of C. Although designed for parallel debugging, the Mantis interface also supports sequential debugging, allowing a single environment for both sequential and parallel debugging.

Mantis currently runs on the Thinking Machines Corp. CM-5 and on networks of workstations and is built using a Tcl/Tk graphical user interface linked to a modified version of the Free Software Foundation's gdb *debugger. Through the application of a clear set of general principles, Mantis has become a practical parallel tool. Mantis made its debut at U. C. Berkeley during the Spring 1994 semester and has been used heavily by the parallel computation course for two years.*

1 Introduction

Parallel programming suffers from a lack of infrastructure. In nearly every aspect of parallelism, systems lean heavily on the attributes of particular classes of applications rather than address more general requirements. Although a diversity of approaches is perhaps the only method of discovering general principles, the lack of consensus prevents the integration of these systems into usable environments. In particular, parallel tools often fail to integrate effectively with other parallel systems, exacerbating the inherent difficulty of successfully programming a massively parallel machine.

In this paper, we present Mantis, a graphical debugger for parallel programs currently running on networks of workstations and the CM-5. Mantis provides integrated support for the Split-C language [2]. Although designed for parallel debugging, the Mantis interface can also be used for sequential debugging in C, C++, Fortran, and other languages, allowing a single environment for both sequential and parallel debugging. Through the application of a clear set of general principles, Mantis has become a practical parallel tool. Mantis made its debut on the CM-5 during the Spring 1994 semester and has been used heavily by the parallel computation course at Berkeley for two years.

Mantis targets the broad class of parallel programs known as bulk synchronous SPMD programs. These programs break into two layers. The top layer consists of a set of bulk synchronous blocks. The processors enter each block simultaneously and synchronize at the end of each block before entering the next. The bottom layer occurs within the blocks. Here the programmer thinks of individual processors—each operates on a different set of data or even with different code, but the model itself is sequential.

Mantis provides support for Split-C, a parallel extension of C. Split-C gives the programmer a clear cost model couched in a small set of abstractions. A simple translation from source to executable code enables high-performance programming. Although Split-C does not require programs to utilize the bulk-synchronous SPMD paradigm, many do. Split-C hosts a variety applications on numerous massively parallel platforms and on networks of workstations.

Successful parallel debugging requires the rethinking of traditional debugging goals in the context of parallelism. Towards that end, we now consider the goals of sequential debuggers. A sequential debugging environment must strive to attain four objectives: first, to support the programmer's conception of the program as defined by the language and programming environment; second, to rapidly focus the user's attention on important data; third, to provide efficient means of performing common tasks; and finally, to be as portable as the program being debugged. Clearly, each objective remains pertinent in the context of parallel debugging. Parallelism expands the range of issues raised by each objective, but does not alter the objectives substantially.

*This material is based upon work supported under a National Science Foundation Presidential Faculty Fellowship CCR-9253705, a Graduate Research Fellowship, and Infrastructure Grant number CDA-8722788, Lawrence Livermore National Laboratories Inst. for Scientific Research Grants #UCB-ERL-92/69 and #UCB-ERL-92/172, and support from Thinking Machines Corporation. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of either organization.

SPDT '96, Philadelphia PA,USA
© 1996 ACM 0-89791-846-0/96/05..$3.50

A further set of parallel objectives can now be drawn from the sequential set. Parallel bugs range a much larger space than do their sequential cousins, making the ability to rapidly focus the user's attention all the more important. Both focusing attention and performing common tasks efficiently require that a parallel debugger use methods that work independent of the actual machine size, *i.e.*, the methods must be scalable. The debugger should also be scalable in terms of program size, enabling the user to debug large, long-running programs. To maintain efficiency on massively parallel systems, the debugger must observe economy of presentation. Finally, portability deserves extra emphasis for parallel systems. These four goals: rapid focus, scalability, economy of presentation, and portability, represent the more difficult aspects of parallel debugging. We discuss Mantis in light of these specific goals.

The remainder of the paper is organized as follows: Section 2 illustrates the interface and functionality through an example debugging session; Section 3 highlights the implementation details of Mantis; Section 4 discusses related work; Section 5 relates user feedback; and Section 6 offers our conclusions.

2 Features

When evaluating a tool, the overall experience of using the tool is often as important as the tool's functionality. For this reason, we feel that Mantis is more naturally illustrated through an example than through abstract descriptions of features. In this section, we step through the process of debugging a parallel program with Mantis. As we encounter new features, we explain their functionality and how they fit into our parallel debugging objectives. At the end, we summarize Mantis in terms of those objectives.

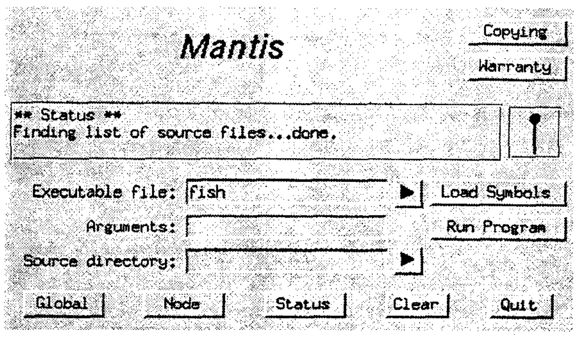
The program to be debugged simulates the world of WaTor [3], which has become a valuable tool for teaching parallel programming at Berkeley. We use a version of WaTor in which fish alone populate an infinite plane and are attracted to other fish according to an inverse-square law. This version introduces programmers to basic issues in data distribution and access as well as typical methods used in solving gravitational and electromagnetic problems.

The solution illustrates the bulk synchronous SPMD programming model discussed earlier. 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. These phases compose the bulk-synchronous layer of the program. Within each phase, the code is sequential, although it operates on the global address space. The program is small but exemplifies the programming model used in many, much larger programs.

Of the two bugs found in this section, one was introduced purposefully from an actual program debugged with Mantis, and the second appeared accidentally while transforming the code into a more legible format.

2.1 Finding a simple bug

The main window is the first to appear in Mantis, as displayed in Figure 1. The window shows the name of the executable file and the arguments to the program, as taken from the command line or entered manually. A third entry box allows the user to override soufce information from



```
** Status **
Finding list of source files...done.
```

Executable file: fish
Arguments:
Source directory:

Figure 1: Main window. Symbols for the `fish` program have just been loaded.

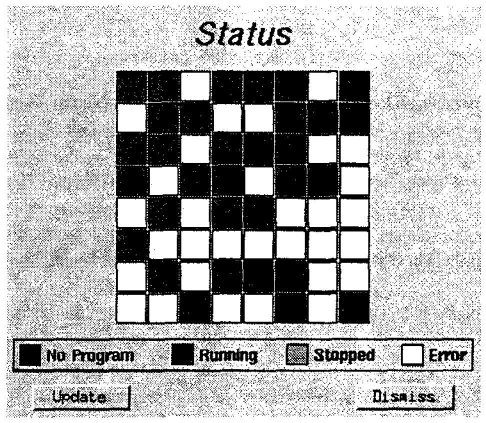


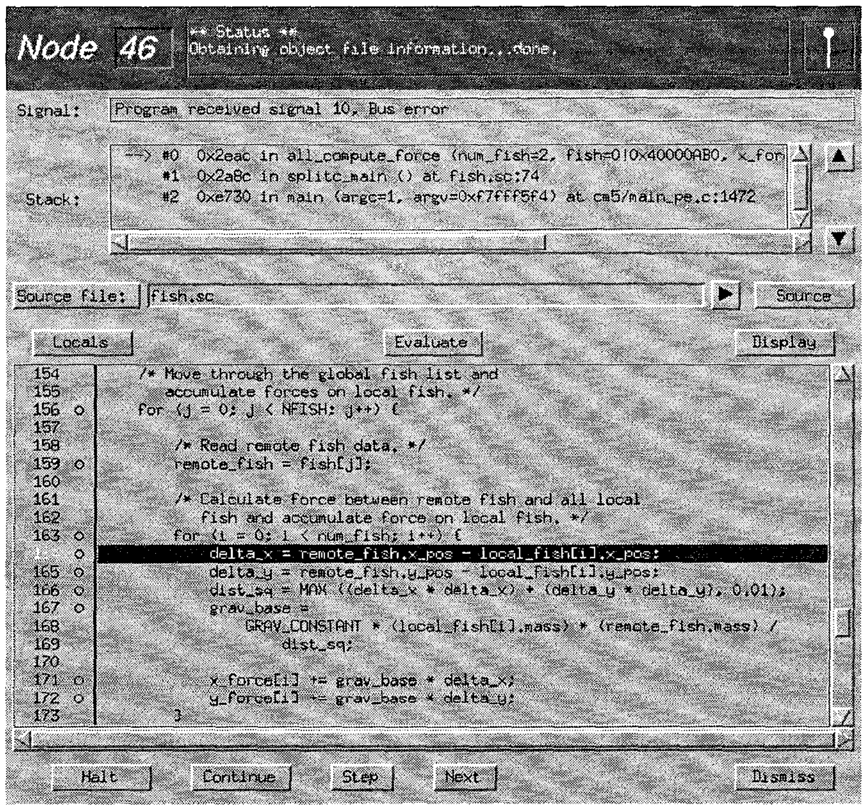
Figure 2: Status window. Many nodes are still running and many others have errors.

the executable. The buttons along the bottom manage the creation of other windows.

The initial version of the program dumped core after encountering a bus error. The first step towards finding the bug is to locate the error—the user runs the program by pressing the `Run Program` button and then opens the status window shown in Figure 2 with another mouse click.

Almost immediately, the bus error occurs. The user detects the error via the status window, which displays the current state of all processors as a two dimensional mesh of squares. The green squares represent executing nodes (perhaps waiting for a reply from another node), and the white squares represent nodes with errors. Clicking on a square creates a node window focused on that processor, allowing the user to detect problems visually and to investigate them rapidly.

Selecting one of the problematic nodes, the user clicks on the square in the status window to create the node window displayed in Figure 3. The node window provides a rich interface to individual processors and serves as the primary means of investigation for the lower layer of bulk synchronous SPMD programs. The entry box at the top of the window indicates the current focus of the window—although Mantis allows multiple node windows, the user can also examine any processor through a single window. Processor numbers correspond to the unique `MYPROC` identifier in Split-C. All status, stack, and display information corresponding to a node window is changed automatically when the user changes the focus.



```
** Status **
Obtaining object file information...done.

Signal:  Program received signal 10, Bus error

Stack:
  --> #0  0x2eac in all_compute_force (num_fish=2, fish=0!0x40000AB0, x_for
      #1  0x2a8c in splitc_main () at fish.sc:74
      #2  0xe730 in main (argc=1, argv=0xf7fff5f4) at cm5/main_pe.c:1472

Source file: fish.sc

154       /* Move through the global fish list and
155          accumulate forces on local fish. */
156 o     for (j = 0; j < NFISH; j++) {
157
158           /* Read remote fish data. */
159 o         remote_fish = fish[j];
160
161           /* Calculate force between remote fish and all local
162              fish and accumulate force on local fish. */
163 o         for (i = 0; i < num_fish; i++) {
    o             delta_x = remote_fish.x_pos - local_fish[i].x_pos;
165 o             delta_y = remote_fish.y_pos - local_fish[i].y_pos;
166 o             dist_sq = MAX ((delta_x * delta_x) + (delta_y * delta_y), 0.01);
167 o             grav_base =
168                   GRAV_CONSTANT * (local_fish[i].mass) * (remote_fish.mass) /
169                       dist_sq;
170
171 o             x_force[i] += grav_base * delta_x;
172 o             y_force[i] += grav_base * delta_y;
173           }
```

Figure 3: Node window. Node number 46 has had a bus error in `all_compute_force` at the line highlighted in black.

The colored bar across the top of the node window reports status information to the user. Each node window uses a distinct color or pattern to provide easy visual identification with the various subwindows corresponding to the node window.[1] Directly below the status region is a region displaying signal and stack information when the processor is halted. Note the content of the signal region in the figure, which confirms the bus error reported earlier. The stack region displays the call stack; the fish program stopped in `all_compute_force`. The user can move between stack frames by pointing and clicking or via a pair of buttons to the right. When the user moves to a new frame, the source file display region changes to show the source code corresponding to the stack frame and highlights the current line in black, as is apparent in the figure.

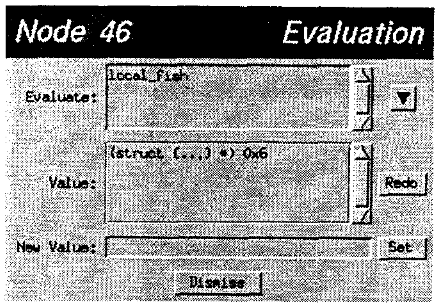
Source code is annotated with line numbers and breakpoint dots. Empty dots represent possible breakpoints, and solid dots represent existing breakpoints. The line of controls just above the source region manages the display. A pull-down menu of three source selection methods appears as a button marked `Source` in Figure 3. "Source" allows the user to select a source file from an alphabetized menu or to view any file by name or dialog. "Function" allows the user to select from a menu of recently examined functions or to view any function by name. "Assembly" is equivalent to the "Function" method except that functions are displayed in assembly code. If "Assembly" is selected when the processor halts or a new stack frame is chosen, Mantis disassembles the function corresponding to the stack frame.

[1]The subwindows are also grouped with the node window so that iconifying or deiconifying the node window has a similar effect on the subwindows.

The left two buttons below the source display area provide a means of halting and continuing the processor. The next two allow the user to step the processor through a line of code—both buttons continue program execution until the next line of code, with the `Step` button stepping into functions and the `Next` button stepping over them. During single-stepping, the pendulum pops up into button form, allowing the user to override possible deadlock.

After looking briefly at the source line at which the error occurred, the user decides to investigate the variables. Examining state in Mantis can be accomplished in several ways. The most commonly used method requires only two clicks of the mouse in the source display region. First, the left button is pressed to highlight 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 variable or expression is highlighted, a right mouse click creates an evaluation window and evaluates the expression. The user selects the expression "local_fish" and brings up the window shown in Figure 4.

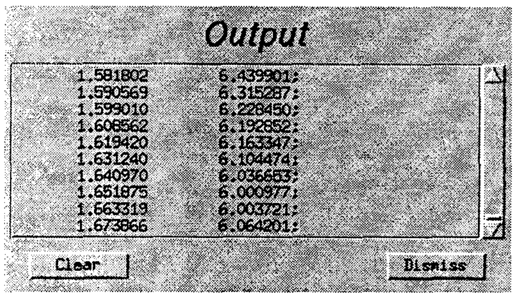
The evaluation window provides a standard interface for evaluating expressions and changing variable values. The top entry box receives the expression, and the middle box returns the value. A menu button to the right of the expression gives a list of recently evaluated expressions, allowing the user to easily cycle through a small set of expressions. The bottom entry allows the user to change the value of an



```
local_fish
```

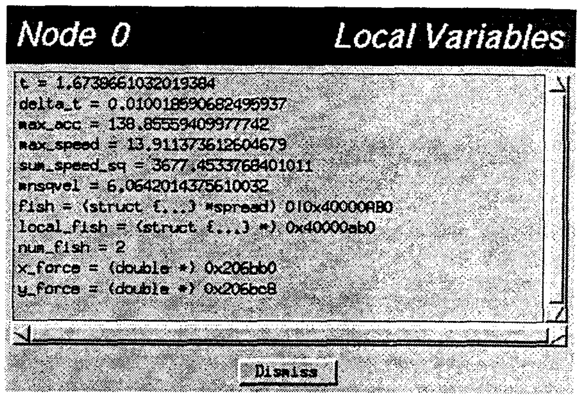
```
(struct {...} *) 0x6
```

Figure 4: Evaluation window. Examining the expression "local_fish" reveals the cause of the problem.



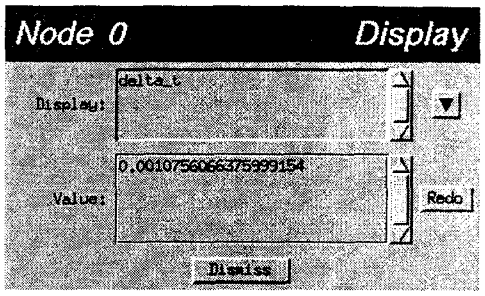
```
    1.581802        6.439901;
    1.590569        6.315287;
    1.599010        6.228450;
    1.608562        6.192852;
    1.619420        6.163347;
    1.631240        6.104474;
    1.640970        6.036653;
    1.651875        6.000977;
    1.663319        6.003721;
    1.673866        6.064201;
```

Figure 5: Output window. The partially debugged program hangs before completion.



```
t = 1.6738661032019384
delta_t = 0.010018590682495937
max_acc = 138.85559409977742
max_speed = 13.911373612604679
sum_speed_sq = 3677.4533768401011
*nsqvel = 6.0642014375610032
fish = (struct {...} *spread) 0!0x40000AB0
local_fish = (struct {...} *) 0x40000ab0
num_fish = 2
x_force = (double *) 0x206bb0
y_force = (double *) 0x206bc8
```

Figure 7: Local variables window. The variables shown correspond to the splitc_main stack frame chosen in the node window.



```
delta_t
```

```
0.0010756066375999154
```

Figure 8: Display window. The expression "delta_t" is evaluated on node 0 each time the processor stops or the user changes the frame.

expression. Errors in evaluation appear in both the node window and in the evaluation window.

Evaluation occurs in the context of the selected stack frame on the halted processor. Mantis fully supports Split-C abstractions, such as the global address space and spread arrays. Global and spread pointers appear as a processor-address pairs: `processor!address`. The binary global pointer creation operator, `!`, is left associative and has precedence between arithmetic operators and logical operators. Creation of global and spread pointers mirrors the language. As with Split-C, local pointers are dereferenced on the processor indicated by the window, regardless of origin.

From the evaluation window, the user learns that the value of "local_fish" has not been initialized. After adding the initialization, the user recompiles the program and tries again.

2.2 Locating a more subtle bug

After fixing the first bug, the user finds that the program hangs after finishing only about a quarter of the time steps. When run with Mantis, the output from the program appears in the window shown in Figure 5. Both `stdout` and `stderr` are channeled into this window, which appears automatically, and user input to the window is delivered to `stdin`. Two buttons at the bottom allow the user to discard the existing output and to dismiss the window completely. In addition to the output from the process, notification that the program has exited or terminated is also given here.

A quick look at the status window reveals that all processors are running, so the user opens a node window (Figure 6) using the `Node` button in the main window (Figure 1). After halting the processor with the `Halt` button, the user sees that the processor has stopped inside of a barrier called from `splitc_main`. A click in the stack region brings up the function, with the barrier call highlighted in black. Switching focus to a few other processors, the user finds that all have stopped in the same barrier.

The problem at this point is fairly clear: one or more nodes have not reached the barrier, and the majority of nodes are idling. To get a better picture of the state on each node, the user opens the local variables window appearing in Figure 7 by pressing the `Locals` button. This window lists variables local to the current stack frame and updates automatically when the frame is changed. In this case, the user sees that the time `t` agrees with the last value shown in the output window, as expected, since node 0 performs the print statements. When the user shifts the focus to another processor, however, the time no longer agrees—somehow the processors have broken the user's concept of bulk synchronous, equal-length time steps. The user hypothesizes that some node has reached `T_FINAL` and believes itself to be done, causing the rest of the nodes to wait indefinitely at the barrier.

To verify the hypothesis, the user sets breakpoints on two nodes at the point in `splitc_main` where `t` is updated. For each node, we open a display window with the expression "delta_t," as shown in Figure 8. The display window is like the evaluation window except that the expression is re-evaluated automatically when the processor halts or the stack frame is changed. Between time steps, the user must merely continue each of the two processors. At the second step, the user notes that the times have diverged.

The user has verified the hypothesis about the nodes becoming unsynchronized, but has yet to understand how this problem occurs. To understand the process, the user shifts the viewpoint to the upper layer of the bulk synchronous paradigm. The global window supports the bulk synchronous view of the Split-C program, allowing the user to toggle breakpoints for all of the nodes simultaneously and to start and stop all of the nodes. After opening the global window shown in Figure 9 with the `Global` button in the main window, the user brings up `splitc_main` and sets a global breakpoint just before the calculation of `delta_t`.

After starting the program, the user waits for all processors to stop, then picks two processors and examines the quantities used to calculate `delta_t`. Finding that both `max_speed` and `max_acc` differ, the user looks up a few lines, realizes that the calls to `all_reduce_to_one_dmax` return different values, and finds the bug: `all_reduce_to_all_dmax`, which returns the reduced value to all of the processors, should have been used instead. The user makes the change and recompiles the program, after which it runs successfully.

2.3 Summary

We now review Mantis in terms of our proposed objectives: rapid focus, scalability, economy of presentation, and portability. The process of finding a bug in Mantis begins with the status window, which visually directs the user to a specific processor or gives a clear indication of phenomena such as deadlock. At the processor level, the user is automatically presented with signal, stack trace, and source information, including highlighting of the current line. Combined with several efficient methods for exploring the state of the program, this information helps the user to focus on bugs rather than on using the debugger. The methods used by Mantis are scalable to hundreds or thousands of nodes without change. The bulk synchronous viewpoint separation helps economy of presentation. Program control issues are handled using either a single processor or all processors simultaneously, requiring only minimal differentiation in presentation. By providing flexibility in window focus, Mantis eliminates the need to juggle between many windows and circumvents scalability problems. Although methods of data visualization can help with the presentation of program state, a commercial package designed specifically for visualization is usually more effective in this regard. Getting data from Mantis to such a package is straightforward. Finally, Mantis gains portability through its relationship with gdb, which is available for nearly all UNIX platforms.

Beyond the specifically parallel goals, a good debugger must also support the programmer's conception of the program and provide efficient means of performing common tasks. The global and node windows allow a Mantis user to view the running program on either level of the bulk synchronous SPMD paradigm, either as a group of processors and set of bulk synchronous blocks or as individual processors and short pieces of sequential code. Split-C abstractions can be used in any evaluation context, and are handled correctly both in input to and output from the debugger. Common tasks such as expression evaluation, program control, and source browsing typically require only a keystroke or a mouse click. When possible, Mantis performs such tasks automatically or shortens the effort required for repetition.

3 Implementation

In this section, we discuss the implementation structure of Mantis and its effects. Mantis consists of a graphical user

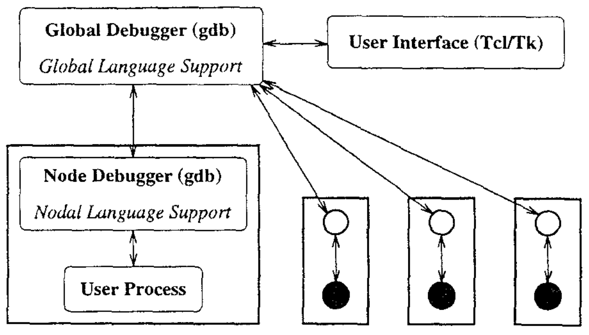


Figure 10: Internal construction of Mantis. Each node debugger uses a standard sequential interface to its child process. The global debugger coordinates node debuggers to provide machine-wide functionality.

interface process that pipes information to and from a debugger child process. The debugger process performs the typical debugging tasks, and the interface presents the information in a more accessible and automatic fashion than that provided by command line debuggers. The internal structure of Mantis appears in Figure 10. In the figure, the bottom four boxes represent processors running a parallel program; each processor also runs a debugger that interacts directly with the corresponding user process. A single global debugger that gathers information from the node debuggers and directs user requests to them. The user interface process communicates only with the global debugger. This structure is designed to minimize the size and breadth of modifications to gdb.

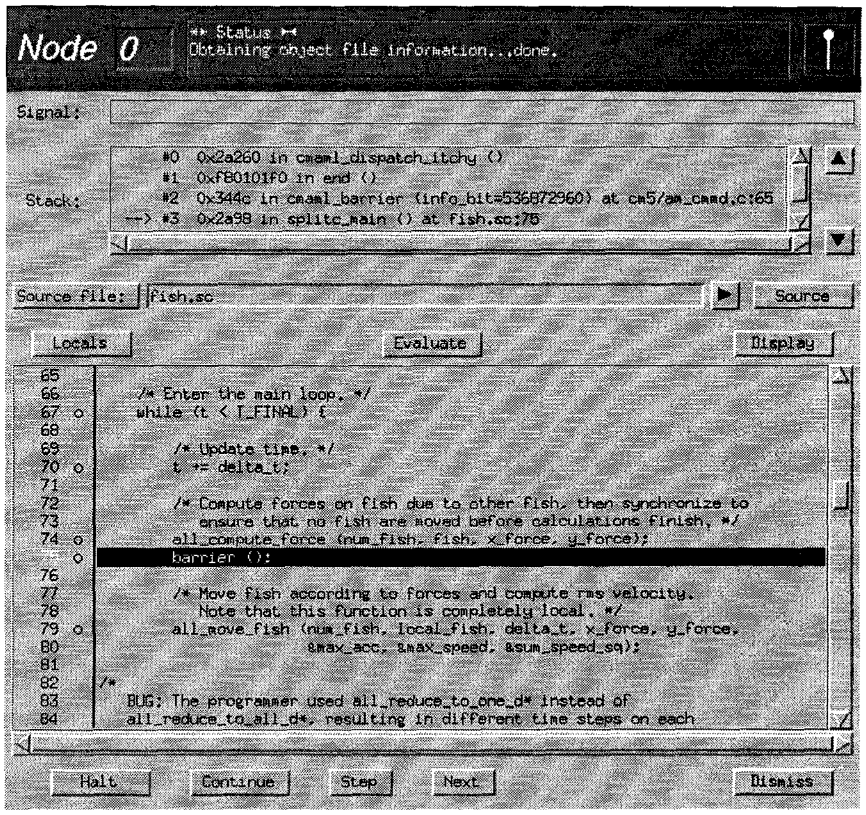
The debugger is based on gdb, the Free Software Foundation's portable sequential debugging environment. Adding language support to gdb required much less time than that required to write a debugger from scratch; a relatively small set of changes transforms a high-quality sequential debugger into a robust parallel debugger. In addition, gdb gives Mantis portability across a wide range of existing and future platforms, as changes are easily incorporated into new releases.

The user interface is based on Tcl/Tk [7], allowing for rapid and flexible creation. For ease of use, Mantis observes known standards in interface design when possible and utilizes common methods when no standards exist. The interface communicates asynchronously with the debugger, allowing it to remain responsive even when the debugger is busy with the user's last request. The interface indicates debugger activity via a rocking pendulum icon and queues actions requiring the debugger until the debugger becomes available.

Like sequential debuggers, Mantis does not perturb program behavior except through direct user intervention, *e.g.*, halting a processor. Neither does Mantis require any changes to an executable file, although compilation with debugging symbols is necessary for source-level debugging. In the absence of source information, Mantis can disassemble the code.

4 Related Work

In this section, we first present alternative methods for parallel debugging, commenting on the advantages and drawbacks of each method. We next examine a handful of parallel debuggers. After noting each debugger's apparent goals, we



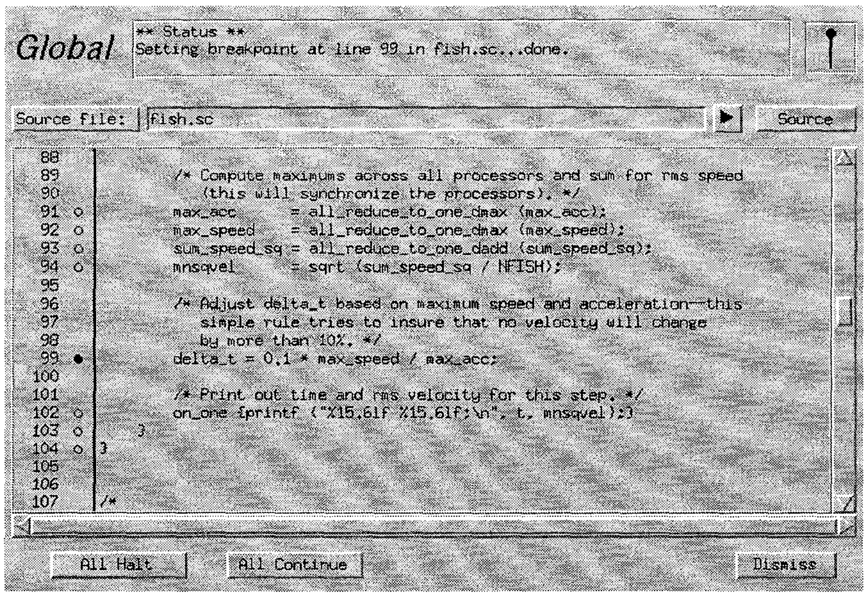
```
Obtaining object file information...done.
```

```
    #0  0x2a260 in cmaml_dispatch_itchy ()
    #1  0xf80101f0 in end ()
    #2  0x344c in cmaml_barrier (info_bit=536872960) at cm5/am_cmmd.c:65
--> #3  0x2a98 in splitc_main () at fish.sc:75
```

```
fish.sc
```

```
65
66         /* Enter the main loop. */
67 o       while (t < T_FINAL) {
68
69             /* Update time. */
70 o           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 o           all_compute_force (num_fish, fish, x_force, y_force);
75 o           barrier ();
76
77             /* Move fish according to forces and compute rms velocity.
78                Note that this function is completely local. */
79 o           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
```

Figure 6: Node window. Node 0 has hung in a barrier. The user has moved up the stack to splitc_main, and the call is highlighted in black.



```
** Status **
Setting breakpoint at line 99 in fish.sc...done.

fish.sc

 88
 89         /* Compute maximums across all processors and sum for rms speed
 90            (this will synchronize the processors). */
 91 o       max_acc      = all_reduce_to_one_dmax (max_acc);
 92 o       max_speed    = all_reduce_to_one_dmax (max_speed);
 93 o       sum_speed_sq = all_reduce_to_one_dadd (sum_speed_sq);
 94 o       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 o       on_one fprintf ("%15.6lf %15.6lf;\n", t, mnsqvel);}
103 o    }
104 o }
105
106
107   /*
```

Figure 9: Global window. The user has just set a breakpoint at line 99 (in splitc_main).

evaluate its capabilities in terms of our objectives in order to facilitate comparison with Mantis. For a more detailed review of these systems, see [4].

4.1 Tracing

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 authors, recognize the potential usefulness but maintain that perturbations to the program and the sheer volume of trace data generated by real programs[2] 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.

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 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 is feasible, the amount of time taken by the tracing code is generally comparable to the faster message-passing calls, possibly perturbing the behavior of the program enough to hide bugs.

[2]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).

4.2 Animation

Program animation generally requires that the user spend time to augment the program with appropriate icons and annotations to allow an animation package to display the results. Animation also entails tracing, often beyond that required for deterministic replay. 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.

4.3 Dynamic instrumentation

The process of dynamic instrumentation involves the addition of short instruction sequences to an existing executable. Parallel performance tuning tools like Paradyn [6] 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 watch points. Future work on Mantis includes the addition of such features along with performance tuning tools.

4.4 Other Parallel Debuggers

Panorama [5] was developed to integrate the myriad techniques developed for parallel debugging in an easily portable and extensible manner. It uses Tcl/Tk to enhance portability and extensibility, translating between its own portable debugging interface and that used by the machine vendor's debugger via a "platform file." Debugging interaction is primarily textual, and visualization interaction is primarily graphical. Visualization windows use an intrusive tracing mechanism, with overhead ranging from 5 to 65% of the cost of a communication call. Although Panorama meets its goals well, the strong emphasis on portability and extensibility tends to exclude other issues. Useful but non-standard features of vendor debuggers are sacrificed in favor of a standard set of commands. The model is also unable to take advantage of important data typically considered internal to debuggers, including source to executable mappings, variable scope, and lists of symbols. The textual interface inhibits the user from rapidly focusing on the problem, and the reliance on vendor debuggers leaves the issues of scalability and economy of presentation open for each system.

Node Prism extends the Prism data parallel debugger for the CM-5 to support the message-passing paradigm [8, 9] and integrates tools for data visualization and performance tuning. Prism addresses scalability by taking advantage of the parallel nature of the debugger itself. Economy of presentation is addressed by compressing any text entering the feedback region into a list of unique responses, again taking advantage of the debugger's parallel nature. A window displaying a tree of stack traces helps to focus the user's attention on any problems in control flow. Data display relies on a single interface with several options. Prism also introduces a flexible method of control that enhances the power of traditional debugging operations, allowing the user to specify arbitrary subsets of processors. Processor sets are named with a variant of data parallel array notation. Despite Prism's success with several of our objectives, it fails to meet the need for rapid focus and even for efficient means of performing common tasks. The main difficulty in this regard is the interface. For example, only one window for viewing source is allowed, and switching between files is slow and cumbersome. Controls are unnecessarily complex and seem to deliberately avoid potential mnemonics. Prism at times runs inordinately slowly at our installation, even when the machine is otherwise unloaded. Combined with sporadic behavior and a complete lack of feedback in some cases, the speed problems result in an interface that can be almost painful to use. Thinking Machines is in the process of making Prism available on a range of platforms; until recently, it was only available on the CM-5.

Originally designed for the BBN Butterfly, the TotalView debugger is now available on a wide range of parallel platforms. TotalView emphasizes simplicity and consistency in its user interface, giving the user a very efficient debugging tool. TotalView had a strong influence on the development of Mantis, particularly on several aspects of the user interface. 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. Several data visualization tools are integrated into the debugger. The TotalView interface caters primarily to the expert user who prefers to use the keyboard over the mouse, making its use somewhat difficult for new users and for those who prefer to use the mouse. Source display windows are tied to individual processes, complicating the task of examining several processors.

Developed approximately a year after Mantis, the Portable Parallel/Distributed Debugger, p2d2, adopts a client-server approach to provide a uniform interface for all platforms, communication libraries, and programming models [1]. By specifying protocols for interaction between a user interface client and a debugger server, p2d2 hopes to separate the development efforts for these two pieces. Additional protocols for communication libraries and HPF preprocessors allow a client to access data normally considered internal to such systems. A prototype based upon gdb and Motif demonstrates the feasibility of the approach. The prototype incorporates status, source browsing, and log messages into a single window. Process status appears hierarchically: one region depicts each process as an icon in a grid of processes, a second region provides a textual explanation for one column of the process grid, and a third region displays stack frames and source code for a single process. The user can customize the icons and conditions used to depict processes in the graphical status region. Textual data display windows permit the user to investigate and alter array sections; the interface routes data to a separate package for graphical visualization. A more detailed review of the prototype client is hard because of the current lack of published documentation. The p2d2 prototype has much in common with Mantis and meets our portability objective in the same manner. Pushing graphical status and source browsing into one window detracts somewhat from scalability, as the user cannot afford to dedicate much space to the status. Gauging its success with other objectives is hard without more details of the interface. As potential standards, the p2d2 protocols are both beneficial and detrimental. The client-server interface embodies concepts derived from decades of experience with sequential debugging, and most new approaches to debugging can be implemented in terms of the primitives provided. The communication protocol, however, is based on PVM and MPI and might tend to exclude alternatives.

5 User Feedback

Since its debut in the Spring 1994 semester, Mantis has been used by the parallel computation course at U. C. Berkeley. Students have favored the integrated Split-C/Mantis programming environment over alternative systems. While we have taken no formal user survey, the feedback we have received has been primarily positive, with a few recurring requests.

One of Mantis' main advantages over the other debuggers available to the students is the ease with which new users can start to use the tool. To our amusement, a myth has arisen that Mantis is the only debugger available. While true in terms of Split-C language support, the myth is otherwise false. The basis for the myth became clear in a conversation with one of the users. After we pointed out that a commercial debugger provided a feature he had suggested, he replied, "Sure, if you can get it to run."

Users praise the use of the status window to focus their attention on errors and the ability to investigate problems with a click of the mouse. They also appreciate the separation of the local and global viewpoints, which mirror most users' understanding of their programs.

The most obvious deficiency in Mantis is its failure to address control flow bugs. Although these bugs have typically been easy to detect, locating the problem can be difficult. Other suggestions have included the addition of library support for the Split-C communication primitives and minor extensions to the user interface.

Until this semester, nearly all use of Mantis was on the CM-5. As students begin to use Mantis with our network of workstations (NOW), we expect several other problems to appear. Scheduling on the NOW is much less deterministic than on the CM-5, making bug reproducibility an issue. Also, although Split-C does not support multithreaded applications, several users have already asked about support for threads with Mantis.

6 Conclusion

The goals of a sequential debugging environment include support for the programmer's conception of the program, rapid focus on important data, efficient means of performing common tasks, and portability. Mantis supports the bulk synchronous and individual node viewpoints comprising the bulk synchronous SPMD paradigm via the global and node windows. Expression evaluation fully supports Split-C abstractions. Automatic source display with highlighting and simple mechanisms for evaluation help the user to focus on bugs rather than on using the debugger. Mantis automates tasks when possible and remembers recent tasks to ease repetition, and the interface remains responsive even when the debugger is busy. Source files are alphabetized into a single menu for quick access. Keyboard shortcuts allow experts to avoid reaching for the mouse, and buttons are available for the keyboard-challenged. Finally, Mantis is as portable as the well-known sequential debugger, gdb.

The parallel objectives also consist of four goals: rapid focus, scalability, economy of presentation, and portability. Mantis uses the status window to rapidly focus the user's attention on bugs at the systemwide level and to drop down to the level of the individual processor. The methods used by Mantis are scalable to hundreds or thousands of nodes without change. Economy of presentation is handled largely through the separation of viewpoints for the two layers of the bulk synchronous SPMD paradigm. Processor control operations use either a single processor or all processors simultaneously. Portability has already been addressed.

Although Mantis has met the goals fairly well, several issues remain. Rapid focus on control flow bugs is largely unaddressed, although adding a version of known methods (e.g., the tree of stack traces in Prism) can adequately handle this problem. The economy of presentation represented by data visualization is also largely absent from Mantis, but visualization is perhaps best handled by those commercial packages specializing in such tasks, and getting the data from Mantis to such a package is straightforward.

The application of a clear set of general principles and the integration with Split-C and the bulk synchronous SPMD paradigm have made Mantis a practical parallel tool. For two years, nearly every student in the parallel computation course at U. C. Berkeley has chosen from a wide range of options to use the integrated Split-C/Mantis programming environment.

Future work involves the addition of dynamic modifications to executable code for fast conditional breakpoints and minor code patches. We are also interested in extending Mantis to include support for performance tuning.

Acknowledgements

Mantis has a long history of encouraging feedback, and we want to thank those people without whose endurance Mantis might never have passed beyond the alpha stage. Specifically, our thanks go to Professor James Demmel and the members of his CS267 classes for putting up with early versions of Mantis, Professor Lawrence Rowe, for his comments on the design of the user interface, and Arvind Krishnamurthy, Seth Goldstein, and Andrea Dusseau, for suggestions on all aspects of the project.

References

[1] D. Cheng, R. Hood, "A Portable Debugger for Parallel and Distributed Programs," Proceedings of Supercomputing '94, Washington, D.C., November 1994, pp. 723-732, available from `www.nas.nasa.gov/NAS/Tools/Projects/P2D2`.

[2] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick, "Parallel Programming in Split-C," Proceedings of Supercomputing '93, Portland, Oregon, November 1993, pp. 262-273, available from `www.cs.berkeley.edu/projects/parallel/castle/split-c`.

[3] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, "Solving Problems on Concurrent Processors," Vol. I, Ch. 17, pp. 307-325, Prentice Hall, Englewood Cliffs, New Jersey.

[4] S. Lumetta, "Mantis: A Debugger for the Split-C Language,", University of California at Berkeley Technical Report #CSD-95-865, January 1995.

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

[6] 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 `www.cs.wisc.edu/~paradyn/papers.html`.

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

[8] 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.

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

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