Mantis User's Guide, Version 1.0

Steven S. Lumetta and David E. Culler *

August 31, 1994

Abstract

This report describes Mantis, a graphical debugger for the Split-C language. Split-C is a parallel extension of C which retains the straightforward translation from source code to executable code necessary for high performance programming of parallel machines. Mantis supports the bulk synchronous and individual node viewpoints which together dominate the design of Split-C programs. Execution can be managed for all nodes as a group or for each node individually. Finally, state and invariants can be checked with a variety of methods, each capable of understanding the abstractions which define Split-C. The graphical interface is simple enough for new users to understand with minimal effort yet powerful enough to allow experienced users to work effectively. Using a straightforward example, we illustrate the process of using Mantis to find both simple and more subtle bugs. We then summarize the important features of Mantis by topic.

Mantis currently runs on the Thinking Machines Corp. CM-5 and is built using a Tcl/Tk graphical user interface linked to a modified version of the Free Software Foundation's `gdb` debugger. Mantis made its debut at U. C. Berkeley during the Spring 1994 semester and was used heavily by the parallel computation course.

*This material is based upon work supported under a National Science Foundation Presidential Faculty Fellowship Award, a Graduate Research Fellowship, and Infrastructure Grant number CDA-8722788, as well as Lawrence Livermore National Laboratories Inst. for Scientific Research Grants #UCB-ERL-92/69 and #UCB-ERL-92/172. 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.

Contents

List of Figures

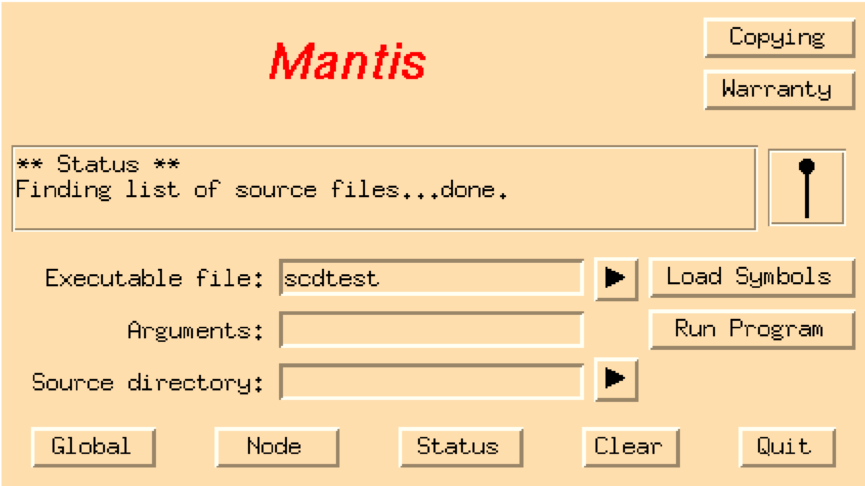

Figure 1: Main window. Symbols for the `scdtest` program have just been loaded.

1 Introduction

This report describes Mantis, a graphical debugger for the Split-C language. Split-C [1] is a parallel extension of C which retains the straightforward translation from source code to executable code necessary for high performance programming of parallel machines. By creating a simple programming model which supports aspects of the shared memory, message passing, and data parallel paradigms, Split-C offers the programmer a clear cost model for programs. Split-C was first implemented on the Thinking Machines Corp. CM-5, building from GCC and Active Messages [8], and the first version of Mantis has been implemented on that same platform. The language has been used extensively as a teaching tool in parallel computing courses and hosts a wide variety of applications; Mantis made its debut during the Spring 1994 semester and was used heavily by the parallel computation course.

The remainder of the report is organized as follows: in Section 2, we discuss the goals of the debugger and the general Split-C programming model; in Section 3, we walk the reader through an example using Mantis to find bugs in a simple piece of code; in Section 4, we summarize the features of Mantis; in Section 5, we explore the implementation details of the debuggers; and in Section 6, we offer our conclusions.

2 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 as given in the compilation environment by allowing the same set

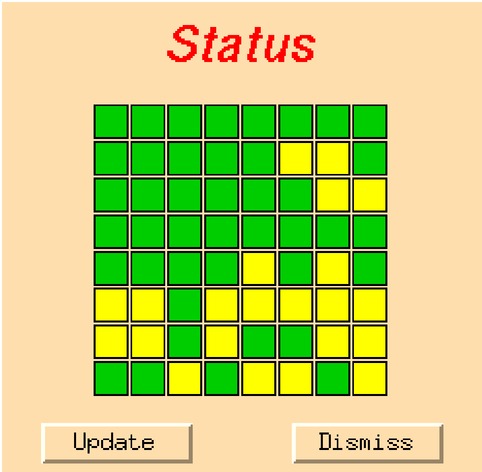

Figure 2: Status window. Many nodes are still running (green/dark) and many others have errors (yellow/light).

of abstractions and the same viewpoint. It must also provide efficient means of performing common tasks such as execution control (e.g., breakpoints) and verification of invariants. What do these mean in terms of Split-C?

In our experience, Split-C programs break into two layers. The top layer consists of a set of bulk synchronous blocks. The nodes enter each block more or less simultaneously and are synchronized at the end of each block before entering the next. The second layer occurs within the blocks. At this layer, the programmer thinks of each node as being distinct—each operates on a different set of data or even different code, but the model itself is sequential.

For the example, we shall draw on code to simulate the world of WaTor, introduced by A. K. Dewdney in 1984 [3] and documented further by Fox et. al. [4], which has become 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 through a small set of rules. For the example, we shall 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. We use this version to introduce programmers to basic issues in data distribution and access as well as typical methods used in solving gravitational and electomagnetic problems.

We present a conceptual solution which illustrates the Split-C programming model discussed above; the commented code appears in Appendix A. At the top level, all processors will 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 since each processor looks at each fish. Although this program is small,

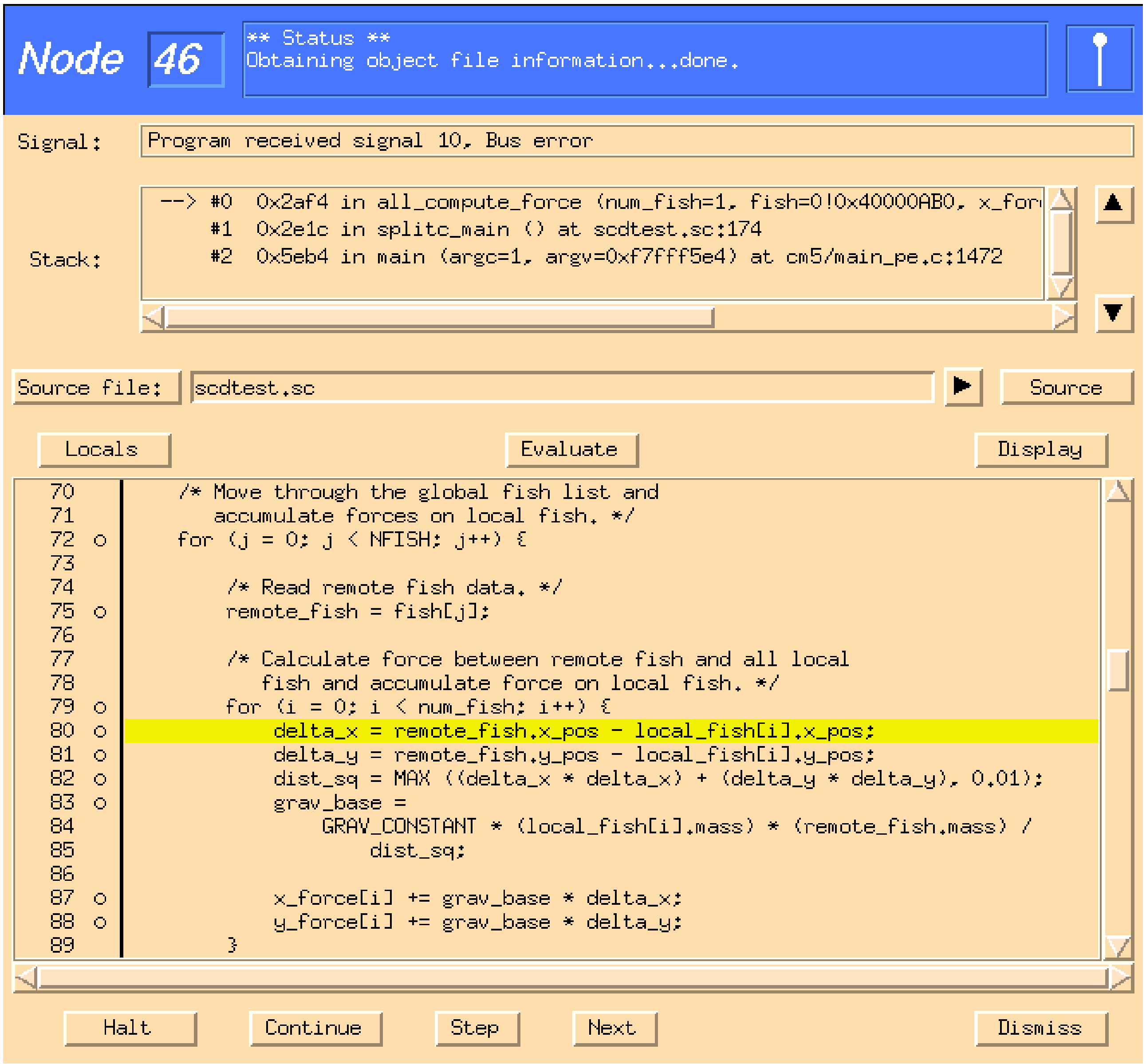

Figure 3: Node window. Node number 46 has had a bus error in `all_compute_force` at the line highlighted in yellow.

it exemplifies the programming model used in many much larger programs.

3 Illustration of Use

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

3.1 Finding a simple bug

We compile and run the program, and it almost immediately encounters a bus error and dumps core, so we start Mantis with the intention of running the program again.[1] After a brief disclaimer dialog, the main window appears, as shown in Figure 1. 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. Since starting a program on the CM-5 takes a noticeable amount of time, Mantis indicates this waiting period to the user through the status area and inverted pendulum icon. The pendulum rocks back and forth until Mantis is ready for another command. The user interface retains full functionality during this period, but any actions which require the attention of Mantis itself are stacked for later execution.

Almost immediately after the example program starts, the bus error occurs. We detect the errors via the status window shown in Figure 2. The green squares (the darker squares in the black and white version) represent nodes which are running (perhaps waiting for a reply from another node), while the yellow squares represent nodes in error. If any of the nodes had stopped at a breakpoint or been halted by the programmer, they would have been displayed in blue.

Picking one of the problematic nodes, we click on the square in the status window to create the node window displayed in Figure 3. The signal section confirms our belief that this is the same problem encountered from the command line, and the stack section shows that the node has stopped in `all_compute_force`. Furthermore, Mantis has focussed the source display section on the top stack frame and highlighted the current line in yellow.

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

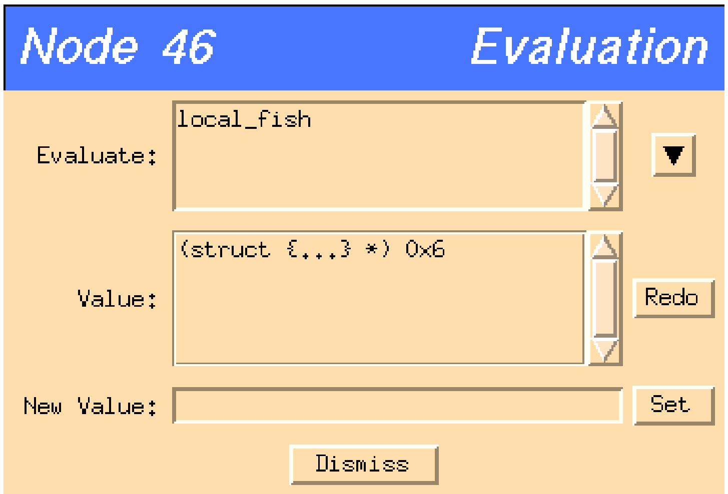

Figure 4: Evaluation window. Examining the expression “local_fish” reveals the cause of the problem.

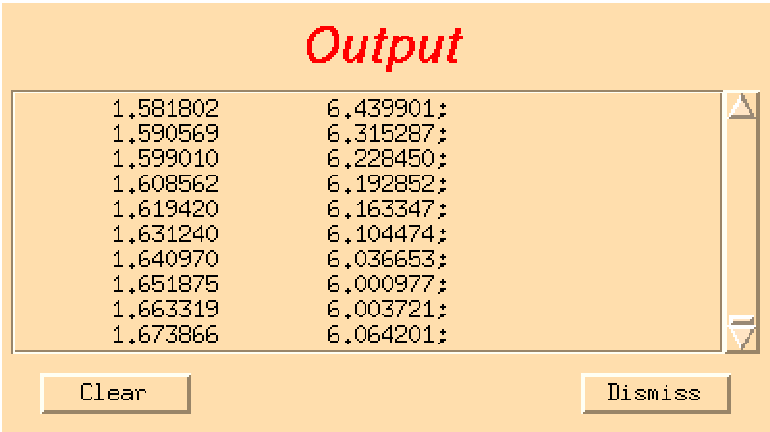

Figure 5: Output window. The partially debugged program hangs before completion.

In addition to these features, the node window controls program execution on a per node basis, supporting the common single-node viewpoint inside of synchronous blocks. The node window also gives access to the various data display windows.

After looking briefly at the source line at which the error occurred, we decide to have a look at the variables used. We select the expression "local_fish" by pointing and clicking the left mouse button, then evaluate it by pressing the right button. The value returns in the window shown in Figure 4. 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. The capabilities include support for the extensions to C such as the global address space and spread arrays. The section at the bottom of the window allows the user to change state by entering a new value for a given variable.

From the window, we learn that the value of "local_fish" has not been initialized (see lines 66-67 in Appendix A). We add the initialization and recompile the program, hoping that we'll have no further problems.

3.2 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 window shown in Figure 5 until finally it stalls. The status window shows that all processors are running.

Using the `Node` button in the main window (Figure 1), we create a node window as shown in Figure 6. We then stop the processor by clicking on the `Halt` button which appears in Figure 6. It stops inside of a barrier, but we see that the stack frame just below the barrier is `splitc_main`, so we click on that frame and arrive at the window shown in Figure 6. Notice that the source file section has automatically displayed the relevant section of code and highlighted the barrier call. 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 all have stopped in 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 figure out why some processors are not going through the barrier. To get a better picture of the state on each node, we open the local variables window appearing in Figure 7 by pressing the `Locals` button. The window shows the values of all variables local to the frame selected in the

[2]There are actually restrictions, primarily function calls and macros, which will be discussed later.

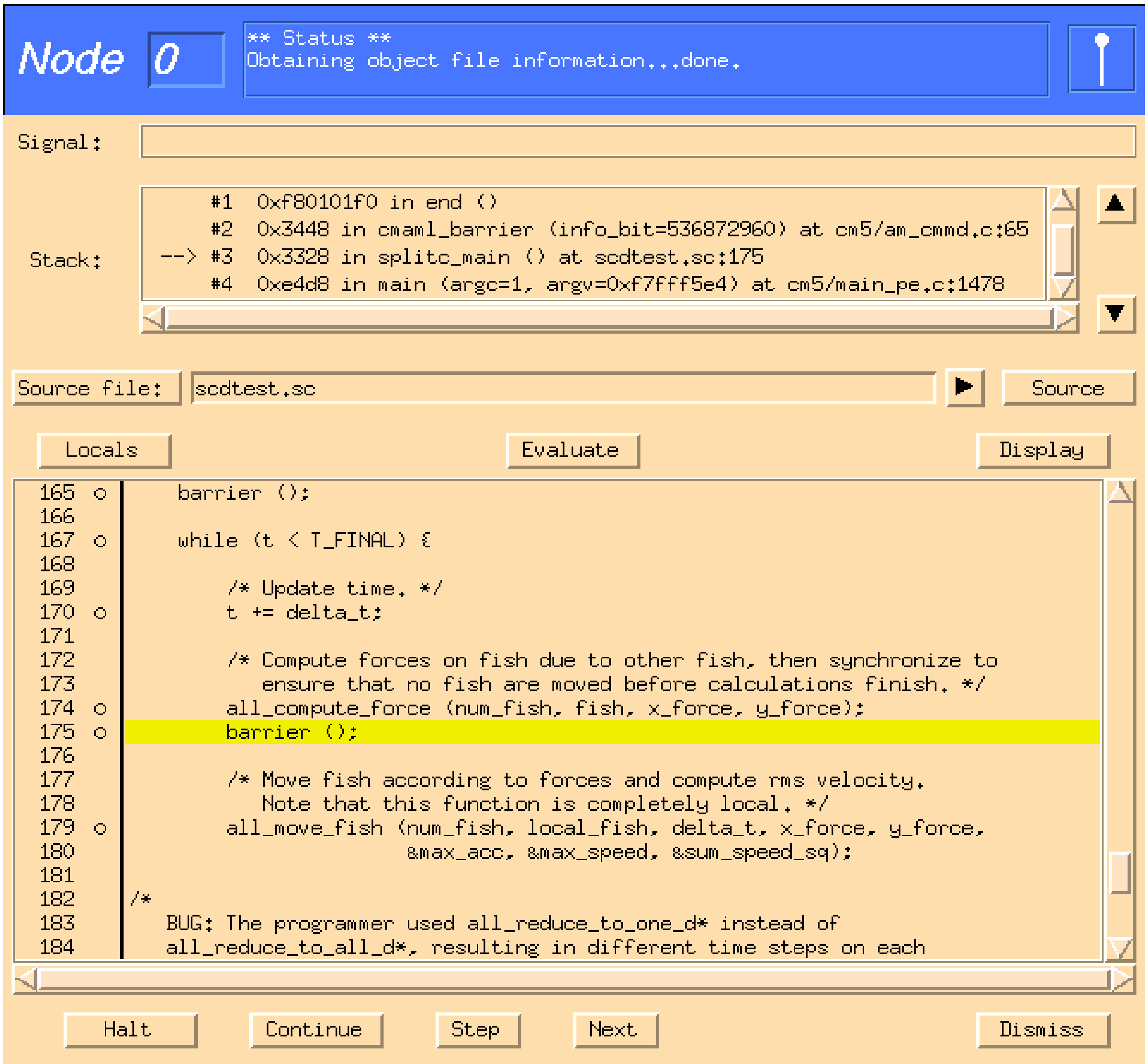

Figure 6: Node window. Node 0 has hung in a barrier. The programmer has moved up the stack to `splitc_main`, and the call is highlighted in yellow.

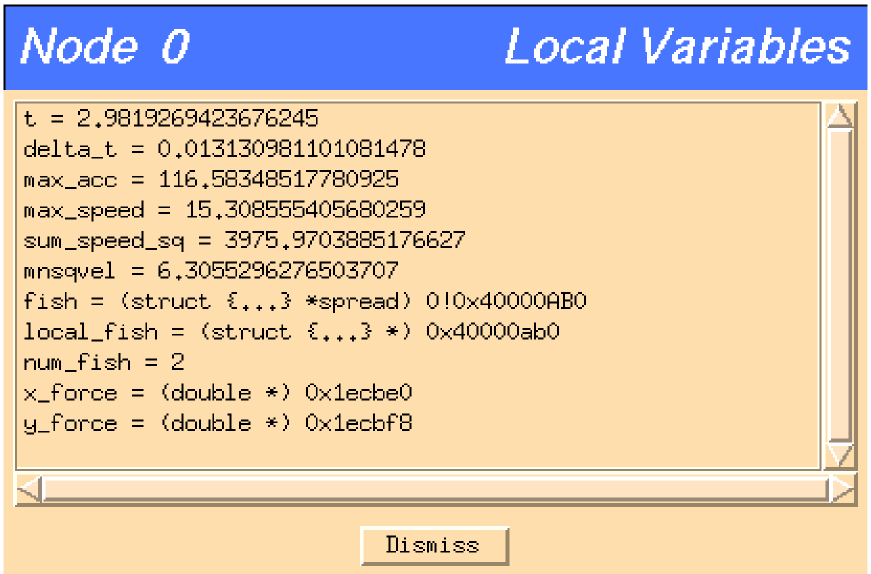

Figure 7: Local variables window. The variables shown correspond to the `splitc_main` stack frame chosen in the node window.

corresponding node window. In this case, we see that the time `t` agrees well 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 have broken the programmer's concept of bulk synchronous, equal-length time steps. We assume 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 this hypothesis, we set breakpoints on two nodes at the point in `splitc_main` where `t` is updated (line number 170). For each node, we open a display window with the expression "delta_t," as shown in Figures 8 and 9. 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. This is ideal for our purposes, because we would like to compare the values of `delta_t` between processors for the same bulk synchronous step, pressing only the `Continue` button of each node window between comparisons. As soon as the second step, we note that the times have diverged, as shown in the figures.

We have verified our hypothesis about the nodes becoming unsynchronized, but we have yet to understand how this occurs. To understand the process, we move upwards for a brief period and consider the bulk synchronous model. We would like to stop all of the processors after they have completed the first few stages; in particular, we would like 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 10.

The global window supports the bulk synchronous view of the Split-C program. It allows

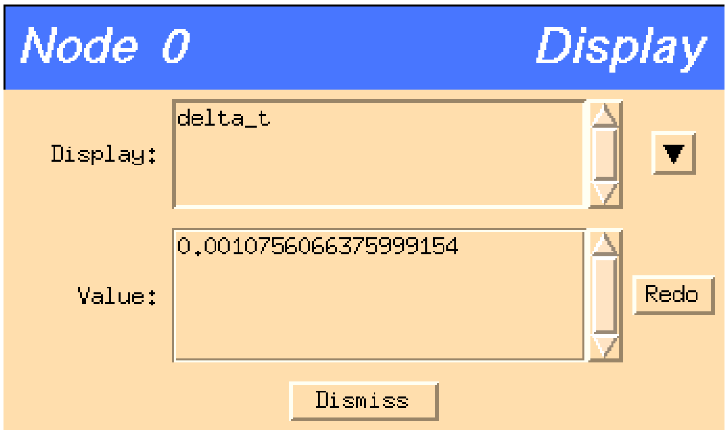

Figure 8: Display window. The expression “delta_t” is evaluated on node 0 each time the processor stops or the programmer changes the frame.

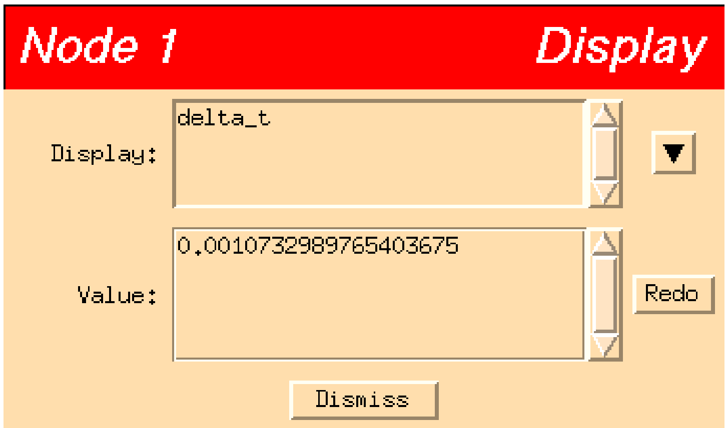

Figure 9: Another display window. The expression “delta_t” is evaluated on node 1.

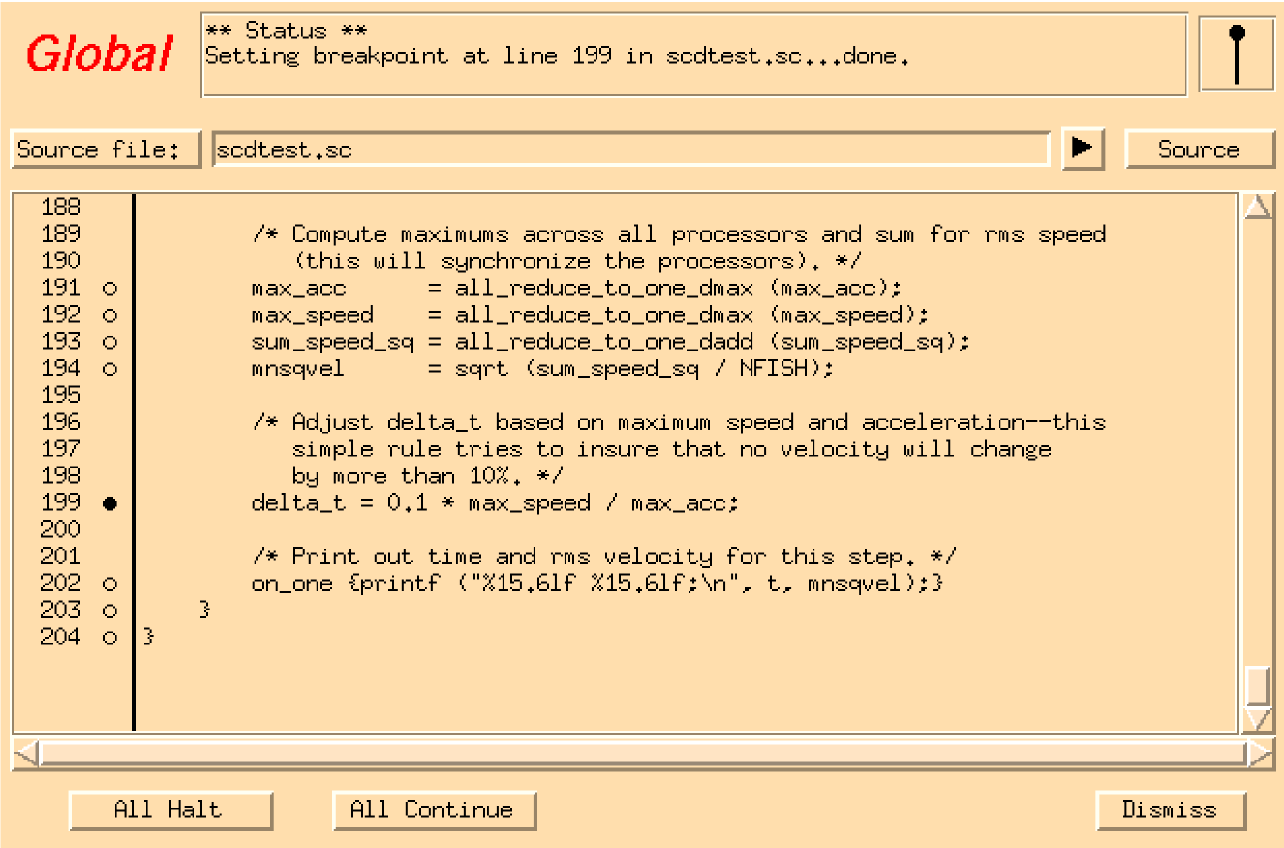

Figure 10: Global window. The programmer has just set a breakpoint at line 199 (in `splitc_main`).

the user to toggle breakpoints for all of the nodes simultaneously, and to start and stop all of the nodes. We set the desired breakpoint, as shown by the solid dot at line 199 in the figure, and then restart the program. Once the processors have all stopped, we can pick two 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 our calls to `all_reduce_to_one_dmax` have returned different values, and we find the bug: we wanted to use `all_reduce_to_all_dmax`, which returns the reduced value to all of the processors (see lines 183-186 in Appendix A). We make the changes and recompile the program, after which it runs as we expect.

4 Summary of Features

In this section, we summarize the features of the Mantis debugger by topic, referring frequently back to the figures used to illustrate the example of use in Section 3.

4.1 Interface highlights

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

- text editing supports a number of standards
  - most commonly used GNU controls (i.e., the `emacs` controls)
  - X cut and paste: left button highlights text, middle button inserts highlighted text
  - PC cut and paste: 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 can be used to move between entry boxes
- the **Return** key can be used in entry boxes to execute an appropriate action; the same action is provided by a control button to the right of the entry
- actions which cause implicit changes in state (e.g., those which kill the process being debugged) request user verification before proceeding

- all windows provide a control button used to dismiss the window; this button is always located in the lower right corner

The debugger process handles requests sequentially and without overlap, but the Mantis interface is built so as to interact asynchronously with the debugger and to enqueue any requests which the debugger is not prepared to handle immediately. The current debugger status is relayed to the user through a status section in each of the major windows (see Figures 1, 3, and 10). This section provides feedback information on commands and errors, while an inverted pendulum to the right rocks back and forth when a command is in progress. Note that not all interface activity requires interaction with the debugger.

Mantis also attempts to make the process of finding source and executable files as simple as possible. Source file information is taken from the executable symbol tables and compiled into a menu of source files.[3] File selection entry boxes also have a control button immediately to the right, always marked with an arrowhead pointing to the right, which allows the user to make choices via a special dialog. The user can traverse directories and select files in this dialog by simply double-clicking with the mouse.

4.2 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 are accomplished directly via the Mantis main window shown in Figure 1, while setting breakpoints typically involves browsing through the source files (see Section 4.3).

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.

The right edge of the main window holds a pair of buttons which provide information on software copying and warranties according to the Free Software Foundation policies.

Along the bottom of the main window is a row of buttons divided into two groups. The right group contains the `Quit` button and a button marked `Clear` which dumps all information and closes all windows.

The left group of buttons manages the creation of other windows. Each of the three

[3]Files in the Split-C library are currently stripped out of this menu; this can also result in user code with the same file name not appearing in the menu. We would like to be able to give a list of header files as well, but this information is not available from the executable.

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

Table 1: Mantis Status Window Colors and Patterns

generates a new window, corresponding to Figures 2, 3, and 10. The global and node windows are described in later sections for the most part, while the status window is discussed here.

The status window (see Figure 2) gives a graphical display of the current state of all processors. The processors are displayed as a two dimensional mesh of squares in a pattern which maximizes the size of each processor within the window size chosen by the user.[4] Table 1 summarizes the colors and patterns used for each possible state (patterns are used only when color is unavailable). Clicking on any of the processor squares brings up a node window corresponding to that processor. Thus if a certain node has caused an error, the user may detect it visually and select it via the mouse for examination.

Halting and continuing the processors can be performed via either a node window (Figure 3) or the global window (Figure 10). In either case, a pair of buttons appears to the left, just below the source display area. In the case of the global window, these buttons will stop or start all processors, while for the node windows, the buttons will stop or start only the processor being examined with that node window. The node window also has buttons for stepping a single processor through a line of code. The `Step` button will continue program execution until it reaches the next line of code or enters a new procedure. The `Next` button will wait for any procedure called to complete and for the processor to reach the next line of code.

Because of the methods employed for single-stepping on Sparc processors, it is necessary for the debugger process to watch each step. Combining this 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 this problem, the pendulum pops up into button form when stepping occurs (instructions also appear in the status area), and if the button is pressed, the processor is halted immediately whether or not it has finished the step.

Output from the process being debugged appears in a special Mantis window (see Figure 5). Both `stdout` and `stderr` are channeled into this window, which appears automatically. Two buttons at the bottom allow the user to discard the data currently in the window and to dismiss the window completely. In addition to the output from the process,

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

notification that the process being debugged has exited or terminated is also given here.

4.3 Source browsing

Source browsing occurs through two Mantis windows: the node window (shown in Figure 3), which focuses on a particular processor, and the global window (shown in Figure 10), which allows interaction with all processors simultaneously. All sources are displayed in the large rectangular region in the center of the windows.

Access to source files and functions is provided by a line of controls just above the source display section. On the right is a menu of selection methods which displays the method currently in use. In Figure 3, this button appears as `Source`. The selection method determines the meaning of the entry box and the button to the left as follows:

Source indicates that anything typed into the entry box will be considered to be a source file. The button to the left of the entry, marked `Source file:` in Figure 3, allows the user to choose from an alphabetized menu of all source files without having to type in the name by hand.

Function indicates that anything typed into the entry box will be considered to be 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) and to display those functions without having to type the name in by hand.

Assembly is equivalent to the **Function** method, except that functions will be displayed as assembly code instead of appearing as the original source.

The source display window is split into two sections. On the left is a static region which displays line numbers and breakpoints. Possible breakpoints are shown with empty circles, and breakpoints which have been set have filled circles. Lines for which no breakpoint can be set have nothing. This region is unaffected by horizontal scrolling. The other section of the source display is simply a line-by-line section of the source file or function selected. The middle mouse button can be used to toggle breakpoints on and off when the associated processor(s) is not running.

4.4 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 3), an entry box shows the processor number and allows the user to examine different processors by simply changing the number and pressing **Return**. All status, stack, and display information corresponding to a node window is changed automatically when the user changes the processor being examined. Node numbers run from 0 to number of processors minus one, 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 which displays signal and stack information when the processor is halted. If a signal (e.g., a segmentation fault) caused the process to halt, that information will be given 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 stack frame, the source file display area changes to show the source code corresponding to the stack frame, complete with a highlighted line where the processor is stopped, as can be seen in Figures 3 and 6.

4.5 Data display and entry

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 simply dragging the mouse with the left button held down. Once the variable or expression has been highlighted, simply pressing the right mouse button creates the evaluation window (shown in Figure 4) 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 then cycle through a small set of expression by simply selecting each from the menu. The bottom entry allows the user to change the value of an expression. Errors in evaluation are shown in both the node window and in the evaluation window to allow the user to focus on either window.

In addition to this common method for checking values, Mantis provides several others. Just above the source display area (see Figure 3) is a row of buttons which manage data display. The `Evaluate` button resides in the middle and is equivalent to pressing the right mouse button is the source display region. The `Display` button to the right creates a window like that shown in Figure 8, which is like the evaluation window except that the expression is re-evaluated automatically when the processor halts or the stack frame is changed. Since 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 which displays variables local to the selected stack frame. Since this information changes with stack frames, it is updated automatically whenever the frame is changed. Note that all data display windows are marked across the top with the color and processor number of the node window to which they correspond. The windows are also grouped with the node window so that iconifying or deiconifying the node window does the same with the subwindows.

Because Split-C currently lacks a standard output format for global and spread pointer values, it was necessary to create one for Mantis. Naturally, Mantis accepts this format on input as well as standard Split-C constructs such as `toglobal` and `tolocal`. Any global or spread pointer 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 argument to the left of the operator is evaluated for processor portion, if it exists, while the value to the right is evaluated for address portion. Thus, it is possible to take two global pointers and form a third using the processor of one and the address of the other. The type of the pointer depends on the type of the second argument. If it has a type, the pointer will be a global pointer to that type. If it has no type, the pointer will be a global pointer to type `void`.

Creation of global and spread pointers mirrors the language. The address of a spread array section, for example, will be a spread pointer if the section consists of one or more of the blocks on each processor, and will be a global pointer if the section is part of a block:

`int a[PROCS*2]::[2];`
`&a[1]` has type `int (*spread)[2]`, while
`&a[1][1]` has type `int *global`.

Casting to a global or spread pointer (or spread array) uses the processor indicated in the window, and casting or creating a pointer with address 0 results in its having processor 0 as well, except in the case of pointer arithmetic.

It is important to keep in mind that local pointers are dereferenced on the processor indicated in the window, regardless of where the pointer was obtained. If, for example, the user has a global pointer to a local pointer to an integer, dereferencing that value twice will return the value at the address of the local pointer in the address space of the processor corresponding to the window, just as it would in Split-C.

Finally, since dynamically allocated arrays appear often in Split-C, it is worth describing 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 will be an array of elements enclosed in braces. Changes can be made by simply typing in a new array, also enclosed in braces, then pressing return (or pressing the set button). Another point of interest is an abbreviation for the fairly common cast-and-dereference occurrence. If, for example, we have a void pointer `ptr` which points to something which we would like to print as being of type `my_type`, we would normally use:

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

But this can be abbreviated as:

```
{my_type}ptr
```

5 Implementation

Mantis is built as a graphical user interface process which 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 most command line debuggers.

The user interface is written using the Tool Command Language (Tcl) [6] and X11 toolkit (Tk) [7] developed by Ousterhout, which greatly simplified the task.[5] The interface required about seven thousand lines of code, which divide roughly equally into script code and C code.

The Mantis debugger is based on the Free Software Foundation's `gdb` debugger. `gdb` consists of roughly two hundred thousand lines of code and provides a portable sequential

[5]It would be hard to overestimate the value of Tcl's interpreted script nature for rapidly creating attractive, functional, and consistent user interfaces.

debugging environment. We felt that the modifications required to add language support and allow parallel processes with `gdb` would require much less time than it would to write a high-quality debugger from scratch. Furthermore, by using `gdb`, we can simply incorporate our changes into future releases and thereby ease portability problems. This has worked to great advantage with Split-C and the `gcc` compiler, and we expect the same results with Mantis and `gdb`.

Unlike Split-C, Mantis currently runs only on the CM-5. Under the CMOST operating system, process information must be obtained through the Time-Sharing Daemon which runs on the CM-5 host processor. To support this model, we use a single debugger process on the host which communicates with the TS-Daemon to gather debugging information. Unfortunately, this means that our debugger process must directly interact with a large number of child processes instead of the usual one-to-one relationship between the debugger and the child. Adding this capability to the gdb debugger was the most difficult modification required, and also the least portable since it permeates a great deal of the `gdb` source. Since most other platforms, in particular networks of workstations, will not require these changes, we have begun to build a more portable version for other platforms and currently plan to finish the new version some time in October 1994. We also hope to make Mantis publicly available by December 1994.

6 Conclusion

How well has Mantis met the goals offered in Section 2? We asserted that the debugging environment must support the programmer's conception of the program as given in the compilation environment by allowing the same set of abstractions and the same viewpoint. Through the global and node windows, Mantis supports the bulk synchronous and individual node viewpoints which together dominate the design of Split-C programs.

We also claimed that a debugger must provide efficient means of performing common tasks such as execution control (e.g., breakpoints) and verification of invariants. In Mantis, execution can be managed for each node individually or for all nodes as a group. State and invariants can be checked with a variety of methods, each capable of understanding the global address space, spread arrays, and other abstractions which define Split-C.

Although Mantis has met the goals fairly well, several issues remain. In general, the problem of condensing information into a format which can be scanned quickly and which directs the user to a bug remains largely unaddressed. Comparisons of stack frames and node-scanning buttons are two ideas which in the future might reduce the amount of information which Mantis users must examine when looking for a bug. Also, it would be useful and fairly straightforward to integrate a data visualization system with the Mantis debugger, but we have yet to do so. Finally, we do not address the ideas of tracing and deterministic replay which currently appear in the literature, primarily because we do not feel that these concepts will be particularly useful in debugging production codes.

During the time in which the first implementation of Mantis was developed, Split-C has been ported to several other platforms [5], including the Intel Paragon and networks of Hewlett-Packard workstations.[6] For those interested in examples of Split-C applications, we suggest [2], which analyzes several sorting codes using the LogP model, and [1], which explains the language by example.

[6]Work is also currently underway to provide Split-C for arbitrary networks of homogeneous workstations as well as the Meiko CS-2.

A Code for Fish and Gravity

```
#include <split-c/split-c.h>
#include <split-c/control.h>
#include <split-c/com.h>
#include <math.h>
#include <malloc.h>


#define NFISH              100      /* number of fish */
#define T_FINAL           10.0      /* simulation end time */
#define GRAV_CONSTANT      1.0      /* proportionality constant of
                                       gravitational interaction */

/*
   This structure holds information for a single fish, including
   position, velocity, and mass.
*/

typedef struct {
    double x_pos, y_pos;
    double x_vel, y_vel;
    double mass;
} fish_t;


/*
    Place fish in their initial positions.
*/

void all_init_fish (int num_fish, fish_t *local_fish)
{
    int i, n;
    double total_fish = PROCS * num_fish;

    for (i = 0, n = MYPROC * num_fish; i < num_fish; i++, n++) {
        local_fish[i].x_pos = (n * 2.0) / total_fish - 1.0;
        local_fish[i].y_pos = 0.0;
        local_fish[i].x_vel = 0.0;
        local_fish[i].y_vel = local_fish[i].x_pos;
        local_fish[i].mass  = 1.0 + n / total_fish;
    }
}


/*
   Compute the force on all local fish according to the 2-dimensional
   gravity rule,
       F = d * (GMm/d^2),
```

```
   and add it to the force vector (fx, fy).  Note that both fish and
   both components of the force vector are local.
*/

void all_compute_force (int num_fish, fish_t *spread fish,
                        double *x_force, double *y_force)
{
    int i, j;
    fish_t *local_fish, remote_fish;
    double delta_x, delta_y, dist_sq, grav_base;

    /* Clear forces on local fish. */
    for (i = 0; i < num_fish; i++) {
        x_force[i] = 0.0;
        y_force[i] = 0.0;
    }

/*
   BUG: The programmer forgot to initialize local_fish before
   using it below.
*/

    /* Move through the global fish list and
       accumulate forces on local fish. */
    for (j = 0; j < NFISH; j++) {

        /* Read remote fish data. */
        remote_fish = fish[j];

        /* Calculate force between remote fish and all local
           fish and accumulate force on local fish. */
        for (i = 0; i < num_fish; i++) {
            delta_x = remote_fish.x_pos - local_fish[i].x_pos;
            delta_y = remote_fish.y_pos - local_fish[i].y_pos;
            dist_sq = MAX ((delta_x * delta_x) + (delta_y * delta_y), 0.01);
            grav_base =
                GRAV_CONSTANT * (local_fish[i].mass) * (remote_fish.mass) /
                    dist_sq;

            x_force[i] += grav_base * delta_x;
            y_force[i] += grav_base * delta_y;
        }
    }
}


/*
    Move fish one time step, updating positions, velocity, and
    acceleration.  Return local computations of maximum acceleration,
    maximum speed, and sum of speeds squared.
*/
```

```
void all_move_fish (int num_fish, fish_t *local_fish, double delta_t,
                    double *x_force, double *y_force,
                    double *max_acc_ptr, double *max_speed_ptr,
                    double *sum_speed_sq_ptr)
{
    int i;
    double x_acc, y_acc, acc, speed, speed_sq;
    double max_acc = 0.0, max_speed = 0.0, sum_speed_sq  = 0.0;

    /* Move fish one at a time and calculate statistics. */
    for (i = 0; i < num_fish; i++) {

        /* Update fish position, calculate acceleration, and update
           velocity. */
        local_fish[i].x_pos += (local_fish[i].x_vel) * delta_t;
        local_fish[i].y_pos += (local_fish[i].y_vel) * delta_t;
        x_acc = x_force[i] / local_fish[i].mass;
        y_acc = y_force[i] / local_fish[i].mass;
        local_fish[i].x_vel += x_acc * delta_t;
        local_fish[i].y_vel += y_acc * delta_t;

        /* Accumulate local max speed, accel and contribution to
           mean square velocity. */
        acc = sqrt (x_acc * x_acc + y_acc * y_acc);
        max_acc = MAX (max_acc, acc);
        speed_sq = (local_fish[i].x_vel) * (local_fish[i].x_vel) +
            (local_fish[i].y_vel) * (local_fish[i].y_vel);
        sum_speed_sq += speed_sq;
        speed = sqrt (speed_sq);
        max_speed = MAX (max_speed, speed);
    }

    /* Return local computation results. */
    *max_acc_ptr      = max_acc;
    *max_speed_ptr    = max_speed;
    *sum_speed_sq_ptr = sum_speed_sq;
}


/*
   Simulate the movement of NFISH fish under gravitational attraction.
*/

splitc_main ()
{
    double t = 0.0, delta_t = 0.01;
    double max_acc, max_speed, sum_speed_sq, mnsqvel;
    fish_t *spread fish, *local_fish;
    int num_fish;
    double *x_force, *y_force;
```

```
        /* Allocate a global spread array for the fish data set and
           obtain a pointer to the local portion of the array.  Then
           find the number of fish owned by this processor. */
        fish       = all_spread_malloc (NFISH, sizeof (fish_t));
        local_fish = tolocal (fish);
        num_fish   = my_elements(NFISH);

        /* Allocate force accumulation arrays. */
        x_force = (double *)malloc (num_fish * sizeof (double));
        y_force = (double *)malloc (num_fish * sizeof (double));

        /* Initialize the fish structures, then synchronize
           to ensure completion. */
        all_init_fish (num_fish, local_fish);
        barrier ();

        while (t < T_FINAL) {

            /* Update time. */
            t += delta_t;

            /* Compute forces on fish due to other fish, then synchronize to
               ensure that no fish are moved before calculations finish. */
            all_compute_force (num_fish, fish, x_force, y_force);
            barrier ();

            /* Move fish according to forces and compute rms velocity.
               Note that this function is completely local. */
            all_move_fish (num_fish, local_fish, delta_t, x_force, y_force,
                           &max_acc, &max_speed, &sum_speed_sq);

    /*
       BUG: The programmer used all_reduce_to_one_d* instead of
       all_reduce_to_all_d*, resulting in different time steps on each
       processor which eventually caused synchronization errors and
       hung the program.
    */

            /* Compute maxima across all processors and sum for rms speed
               (this will synchronize the processors). */
            max_acc      = all_reduce_to_one_dmax (max_acc);
            max_speed    = all_reduce_to_one_dmax (max_speed);
            sum_speed_sq = all_reduce_to_one_dadd (sum_speed_sq);
            mnsqvel      = sqrt (sum_speed_sq / NFISH);

            /* Adjust delta_t based on maximum speed and acceleration--this
               simple rule tries to insure that no velocity will change
               by more than 10%. */
            delta_t = 0.1 * max_speed / max_acc;

```

```
            /* Print out time and rms velocity for this step. */
            on_one {printf ("%15.6lf %15.6lf;\n", t, mnsqvel);}
        }
    }
```

References

[1] 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, Portland, Oregon, November 1993, pp. 262-273.

[2] D. E. Culler, A. C. Dusseau, K. E. Schauser, R. P. Martin, "Fast Parallel Sorting under LogP: from Theory to Practice," in "Portability and Performance for Parallel Processing," A. J. G. Hey and J. Ferrante, eds., pp. 71-98, John Wiley & Sons, Ltd., 1994.

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

[4] 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.

[5] S. Luna, "Implementing an Efficient Portable Global Memory Layer on Distributed Memory Multiprocessors," U. C. Berkeley Technical Report #CSD-94-810, May 1994.

[6] J. K. Ousterhout, "Tcl: An Embeddable Command Language," *Proc. USENIX Winter Conference*, pp. 133-146, 1990.

[7] J. K. Ousterhout, "An X11 Toolkit Based on the Tcl Language," *Proc. USENIX Winter Conference*, pp. 105-115, 1991.

[8] 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