Efficient Tracing for On-the-Fly Space-Time Displays in a Debugger for Message Passing Programs*

Robert Hood and Gregory Matthews
Computer Sciences Corporation – NASA Ames Research Center
{rhood,gmatthew}@nas.nasa.gov

Abstract

In this work we describe the implementation of a practical mechanism for collecting and displaying trace information in a debugger for message passing programs. We introduce a trace format that is highly compressible while still providing information adequate for debugging purposes. We make the mechanism convenient for users to access by incorporating the trace collection in a set of wrappers for the MPI communication library. We implement several debugger operations that use the trace display: consistent stoplines, undo, and rollback. They all are implemented using controlled replay, which executes at full speed in target processes until the appropriate position in the computation is reached. They provide convenient mechanisms for getting to places in the execution where the full power of a state-based debugger can be brought to bear on isolating communication errors.

1 Introduction

Software developers who need to debug message-passing programs are usually forced to work at a low level of abstraction. This is especially true when it comes to isolating errors in the interprocess communications. Even if a distributed debugger is available, there are rarely any high-level features for dealing with messages.

One way to address this problem is to provide the user with both a big picture of what has happened during program execution and a way of using that picture to steer the debugging session. For example, it would be a considerable improvement for debuggers to have a space-time diagram [7] that

- shows a timeline of the events, such as messages, that have occurred in each process so far in the computation, and
- has a replay mechanism that is sensitive to "breakpoints" inserted in the timeline.

Previous papers have identified the advantages of having an abstract view of the execution history or of having a replay mechanism available for distributed debugging. A 1998 paper by Frumkin, Hood, and Lopez [3] presented a clear, consistent view of high-level message traffic debugging features based on collection and visualization of program traces and a debugger-managed replay mechanism. In this paper we make that vision practical by

- making trace collection convenient for the user,
- greatly reducing the size of the trace data,
- making the connection between trace collection and viewing tighter, and
- introducing new debugging operations based on the trace display.

In the rest of this paper we describe our approach to making trace-driven debugging practical. We begin by discussing the requirements on the trace information and how to make its collection convenient and efficient. In section 3 we describe the user interface. Following that we give an extended example. In section 5 we detail our experiences with the implementation, including a discussion of the space efficiency of the trace. We then discuss related work and draw conclusions.

2 The Trace

In collecting trace data to be used during a debugging session, there are several requirements to be met.

Information collected: There must be enough information collected to provide the user with an abstract picture of what has happened during the computation. At a minimum for message passing programs, this means a record of each interprocess communication.

Convenience: Data acquisition must be very simple for the user. In particular, there should be minimal changes required of the compilation process. In addition, there shouldn't be any differences between the program source seen in the debugger and the one the user maintains.

Size of trace: In order to be effective, the size of the trace must be manageable. If it is small enough, the trace may be able to stay in memory, thus avoiding the expense of accessing secondary storage.

Availability to trace viewer: If the debugger is going to present an up-to-date picture of the execution so far, it must have access to all of the trace data that has been recorded during execution. In particular, it cannot wait until program execution has terminated and the trace data across all processes has been merged and put in a canonical form. Instead, it must be able to access each process's trace data independently—getting it out of memory if necessary.

*This work was supported through NASA contracts NAS 2-14303 and DTTS59-99-D-00437/A61812D.

It is important to note that trace timestamp accuracy is *not* required. In a debugging session, the relative ordering of events is important; knowing the exact amount of time between two events is much less important.

2.1 Collecting Options

There are several trace collection options to be considered with regard to these requirements.

Source-to-source instrumentation: In this approach, instrumentation is inserted automatically at the source level of the program, which must then be recompiled. One problem with this is that the user's build procedure (e.g the program's Makefile on a Unix system) must be modified to reflect the instrumentation step. Furthermore, the "source code" that the debugger has symbols for is not the same one that the user wants to work with. An additional disadvantage of this method is that the instrumentation points must be identified before execution.

Object code instrumentation: In this method there is a modification step performed on an object code representation—anywhere from the assembly language level down to a fully linked executable. With this approach we can eliminate the problem of having multiple versions of the source. However, we still have the problems that the build procedure must be modified to include the object code instrumentation pass. In addition, the instrumentation decisions must again be made before execution.

Dynamic instrumentation: With dynamic instrumentation, such as that provided by the Dyninst API [2], a running executable is modified using a sophisticated patching technique. This approach overcomes the restriction that instrumentation decisions be made before execution. The need to modify the user's build procedure is also removed. The chief problem with this approach is that the modifications made by the dynamic instrumentation are incompatible with control by a conventional debugger such as *gdb* from the Free Software Foundation. For example, the debugger has no symbol table information for the patches that are inserted in the code and if execution stops in a patch, the debugger will be unable to show the user where execution has stopped.

Wrapped calls to communication libraries: In this approach, the user's code calls a traced version of routines from the message-passing library. In the case of MPI [9], the profiling interface permits this to be done without modifying the source code. From the user convenience perspective, this approach requires only that the link command in the build procedure be changed to use a debugging version of the MPI library. As with the other static instrumentation techniques, we are required to make instrumentation decisions before execution.

For our work we chose to use an instrumented version of the MPI library. We felt that its impact on the compilation process (i.e., one small change to the build procedure) was acceptable, since debugger users usually need to add a compiler flag such as "-g" anyway[1]. We also felt that the flexibility provided by dynamic instrumentation in this case did not outweigh the problems that its incompatibilities with *gdb* would cause in our prototype implementation.

2.2 Tracing with compressibility in mind

One of the goals of this work is to reduce the size of the trace that is collected. An obvious strategy for size reduction is to incorporate compression during collection. While compressing raw trace data is a start, we also modify the trace records to improve the potential for compression. In particular, we want to make it possible to have conventional string compression routines find the patterns of events that often exist in scientific programs.

In order to make patterns apparent, we eliminate the absolute time stamp that normally appears in each record. While a trace without timestamps would contain all of the causal information needed for debugging, the user is better served with an approximation of the time of an event. Without the approximation, there might be little correspondence between the displayed trace and the user's conception of what happens during execution.

It is sufficient for display purposes to approximate absolute timestamps from the data in the trace. So in the interest of compressibility, we stamp each trace record with an integer approximation of the amount of time since the last record. With a suitably coarse approximation, we expect to see a large number of repeated records in the trace. These would then be fodder for the data compression.

For the coarse approximation of the time delta, we use the integer part of the logarithm (base 2) of the number of ticks (i.e., the number of time quanta as reported by some timing source such as a system call or `MPI_Wtime`) since the last record. Basically, we represent a delta D, by:

(number of bits in the binary representation of D) − 1 .

Consider the following example. In a four process execution of a 2-D implicit flow solver using a so-called multipartition domain decomposition strategy, the communication pattern for process 0 has three phases during each time step:

neighbor-exchange; X-sweep; Y-sweep

The *neighbor-exchange* phase generates the following events. (The labels are for illustration purposes only.)

```
N1:   POST_RECV source=1
N2:   POST_RECV source=3
N3:   SEND dest=1
N4:   SEND dest=3
N5:   WAIT source=1
N6:   WAIT source=3
        [computation]
```

[1] To be truly convenient to the user, when the compiler gets a "-g" on the link command, it should use the debugging version of the MPI library.

where the "*[computation]*" indicates substantial computation. The *X-sweep* phase generates:

```
            [computation]
X1:     SEND dest=3
X2:     POST_RECV source=1
X3:     WAIT source=1
            [computation]
X4:     SEND dest=3
X5:     POST_RECV source=1
X6:     WAIT source=1
            [computation]
X7:     SEND dest=3
X8:     POST_RECV source=1
X9:     WAIT source=1
            [computation]
X10:    SEND dest=1
X11:    POST_RECV source=3
X12:    WAIT source=3
            [computation]
X13:    SEND dest=1
X14:    POST_RECV source=3
X15:    WAIT source=3
            [computation]
X16:    SEND dest=1
X17:    POST_RECV source=3
X18:    WAIT source=3
            [computation]
```

The *Y-sweep* phase generates the same events as the *X-sweep*, except that processes 1 and 3 are interchanged.

When these events are time-stamped with the approximate time deltas, many of the records are likely to be identical, such as X2=X5=X8. Not only will individual records be duplicated, it is also likely that sequences of records will repeat, such as (X4-X6)=(X7-X9). With a suitably coarse time approximation, it may even be the case that the records for an entire time step of the simulation will be repeated in the next time step. These repetitions, at any level, contribute to compressibility.

One concern that we had with using timestamp approximations was that the errors would propagate. For example, if several records in a row underestimated the time delta, we could end up with a reconstructed absolute time that bore little resemblance to the real time. A simple adaptation of our strategy fixes this problem. Rather than approximating the number of ticks since the absolute time of the last trace record, we approximate the number of ticks since the *reconstructed* time of the last record. With this approach, rather than accumulating, the errors tend to get corrected over time.

2.3 Compressing strategy

Trace records will be created by each call to the wrapped message-passing library, and, ideally, each record will be compressed at the time of creation to minimize the amount of uncompressed trace data residing in memory. This requires the use of a compression library that supports frequent in-memory compression of relatively small amounts of data. The *Zlib* compression library [18] was chosen for its direct support of these desirable capabilities, and for its availability on most modern operating systems. *Zlib* basically allows in-memory use of the compression algorithm and routines used by the Free Software Foundation's *gzip*.

Zlib maintains internal buffers of compressed data that are augmented with each subsequent compression call. These buffers are not suitable for decompression and use until a call is made to flush them, performing some final compression/preparation steps on the buffer. Performing a flush degrades the potential compression of the overall data stream, so that minimizing the frequency of trace data decompression and use by the viewer also minimizes the compressed size of the trace.

The choice of how often to refresh the user's view of the trace dictates the frequency of trace flushes, and thus the degree to which the trace can be compressed. Refreshing after each message exchange may be helpful to the user, but is more suited to the animation features of post-mortem trace visualizers [16]. Refreshing at each process interruption (user defined breakpoint, step in execution, etc.) is a much better choice, for two reasons.

- Users can dictate when the trace view is refreshed by setting breakpoints, and in most cases this will not involve setting a breakpoint after each message exchange in the program being debugged. This reduces the memory footprint of the collected trace.
- Presumably a user defining a breakpoint or performing a step in execution has a rough idea of what message exchanges will take place up to that point in the execution (and can determine more exactly from the source). This reduces the need for a real-time display of messaging.

The degree to which the trace can be compressed is therefore proportional to the ratio of the number of process interruptions over the number of wrapped message library calls. Assuming the user utilizes the trace viewer as a high-level debugging tool, giving a small ratio of process interruptions to wrapped message library calls, this proportionality allows better compression in lengthy executions of message-intensive programs where increased compression is desired.

3 The User Interface / The Viewer

A trace viewer that is part of a distributed debugger must provide more functionality than one that is part of a post-mortem performance analysis tool. In particular, there must be a tight connection between the trace collection and its viewing. There are two related aspects to this.

- Traces being viewed may be incomplete. That is, program execution will not yet have finished.
- The trace creation will be incremental. That is, a partial trace will be displayed. When execution is started and stopped again in the debugger there may be additional records to show in the display.

A further requirement of the trace view is that there be a connection between the viewer and the debugger in order to initiate replay operations. For example, it should be possible for the user to mark a point in the space-time display and have the debugger re-execute the program, stopping at the indicated place in the computation.

With these goals in mind, we implemented a trace viewer in the Portable Parallel/Distributed Debugger (*p2d2*), a debugger for message-passing programs that has been developed at the NASA Ames Research Center. In the rest of this section, we describe that implementation.

3.1 Reading / reconstructing the trace

Each message passing process in a *p2d2* debugging session has its own address space, copy of the wrapped message passing library, and *gdb* debugger process controlling it through which *p2d2* debugging commands are issued [5]. The trace compression described in section 2.3 is performed on a per-process basis, and requires that *p2d2* obtain the trace data from each process in order to display it to the user.

The *gdb* debugger process allows *p2d2* to initiate function calls in the target process while it is stopped at a breakpoint. Functions are included in the wrapped message-passing library to flush the internal buffers maintained by *Zlib*, and allow *p2d2* access to the finished compressed data through further *gdb* data retrieval commands. This compressed data is then decompressed and processed by *p2d2* in its own address space.

P2d2 processes each trace buffer sequentially, with no requirement on ordering, avoiding the added complexity of collating or otherwise pre-analyzing trace buffers in any way. Extracting the execution history from these per-process trace buffers therefore requires that *p2d2* support incremental buildup of the execution history. Incremental buildup places no debugger-based limitation on the use of breakpoints in message passing programs, since process interruptions may occur when messages are partially complete. Thus the execution history stored by *p2d2* may contain incomplete messages—individual sends, receives or waits without corresponding operations on the other side of the message.

3.2 User interface operations

Once the trace has been read by the debugger, it is displayed as a space-time diagram. Messages are displayed as diagonal lines from the sending to the receiving process. With this display the user can see the whole trace or zoom in on a portion of it. If zoomed, the trace can be panned forward or backward in time.

There are three ways to steer the debugging session from the trace, using a form of program re-execution where the computation is halted at an appropriate place in the timeline. We discuss each of these mechanisms in turn.

Stopline: The first form of controlled replay is for the user to mark a single point in the space-time display—indicating one moment in the computation history of a single process. When the execution is started over, the selected process will be halted at the marked event. The user can specify where the other processes stop:

- either at the last event where they could possibly affect the values in the selected process at its stopping point, or
- at the first event where their own values might be affected by the values in the selected process where it is stopped.

In effect, the user is putting a breakpoint in the timeline for one process and having the others stop as soon or as late as possible, consistent with honoring data flow dependences.

Undo: The second form of controlled replay has the effect of undoing a process control operation in the debugger. For example, if the user requested a CONTINUE rather than a STEP, execution might proceed past a critical point that needs to be investigated. The UNDO operation would re-execute the program and stop at a point close-to, but not past, the place where execution was before the CONTINUE was mistakenly issued.

Rollback: The third form of controlled replay mimics the effect of reversible execution. The intention of this operation is to stop the re-execution one message-passing event before its current point. What the user sees is that execution has been rolled back a small amount—in effect reversing it some.

3.3 Implementation of controlled replay

There are two aspects of the controlled replay that we need in order to implement the operations of the previous section:

- ensuring an equivalent execution in the case of nondeterministic programs, and
- stopping the target processes at the right place.

In this work we have assumed that the subject program is deterministic and have concentrated on the second issue. In future work we will integrate a deterministic replay scheme [12] (i.e., one that honors the event orderings from an instrumented run) into our prototype so that we can accommodate nondeterministic programs.

In order to stop at the right place in a re-execution in an efficient way, we need to have the target processes themselves determine when they should stop. To do this, the debugger and the target processes must be in agreement about how the timeline is labeled and each process must be able to compute the label of its current location on the timeline. Then, when execution is restarted, the debugger can load the label of the desired

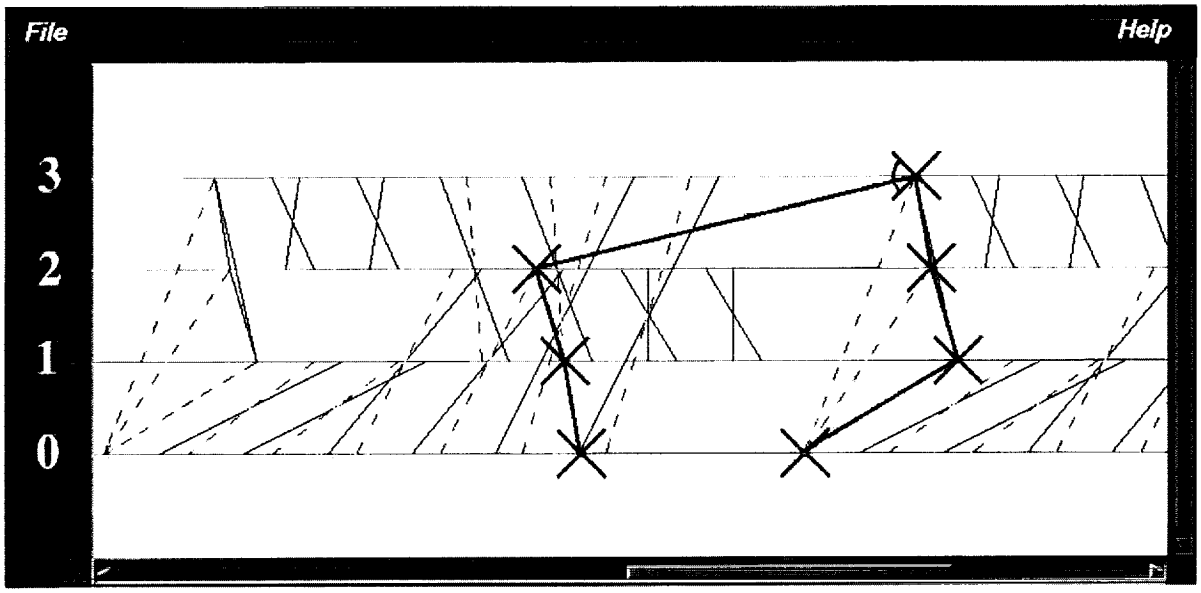

Figure 1: Partial history displayed by *p2d2* for NPB [12] program BT. Dashed lines indicate a message that appears to be traveling backwards in time (see discussion in Section 5.1), and bold lines and large X's indicate events along the past and future frontier (see section 3.4). BT follows a *neighbor-exchange; X-sweep; Y-sweep; Z-sweep* communication pattern for each time step, very similar to (but not exactly the same as) the pattern described in section 2.2. Displayed is one entire time step followed by approximately half of the next. The user has chosen a concurrency point in process 3 that will stop the processes at the beginning of the latter time step, using the future frontier. (The selected concurrency point is denoted by the arc on the left-hand side of the X on the top line).

stop location on the timeline into each process and have it proceed in the computation. When a process computes that it is at the desired stopping point, it can force the execution of a breakpoint trap instruction.

3.4 Calculation of the stopline frontiers

In the 1998 paper by Frumkin, Hood, and Lopez [3] the notion of a *consistent frontier* was used to find a set of events in which no event provably occurs before another. A user could choose an event, either a message send or receive in the displayed execution history, to be the *concurrency point* from which frontiers could be found. Past and future frontiers were then displayed for events that were guaranteed to be in the past or the future, respectively, relative to the chosen event. These frontiers are illustrated in the time-space display shown in Figure 1.

We have extended this capability by providing the user with the option of viewing past and future frontiers relative to *just before* or *just after* the event occurs. Choosing a time in the execution history just before a message send event displays a future frontier that includes the receive event of that message. Choosing a time just after the send event displays a future frontier with events guaranteed to be in the future beyond the corresponding receive event. The past frontier for both these examples would be the same, though.

Choosing a time in the execution history just after a message receive event displays a past frontier that includes the send event of that message. Choosing a time just before the receive event displays a past frontier with events guaranteed to be in the past of the corresponding send event. The future frontier for both these examples would be the same.

4 An Example Usage Scenario

In order to illustrate how trace-driven debugging appears to a user, consider the following scenario. The user is experiencing problems with an MPI program that iteratively solves for the vector X in the matrix equation $A * X = B$. The program seems to hang indefinitely, with no warning messages or errors produced by MPI.

The program breaks down into two main phases: a workload distribution phase where the rows of the matrix A are split up among the processes, and a computation phase where values for the next iteration of the X vector are found at each process and then collected together by a master process (process 0), which then distributes X for the next iteration.

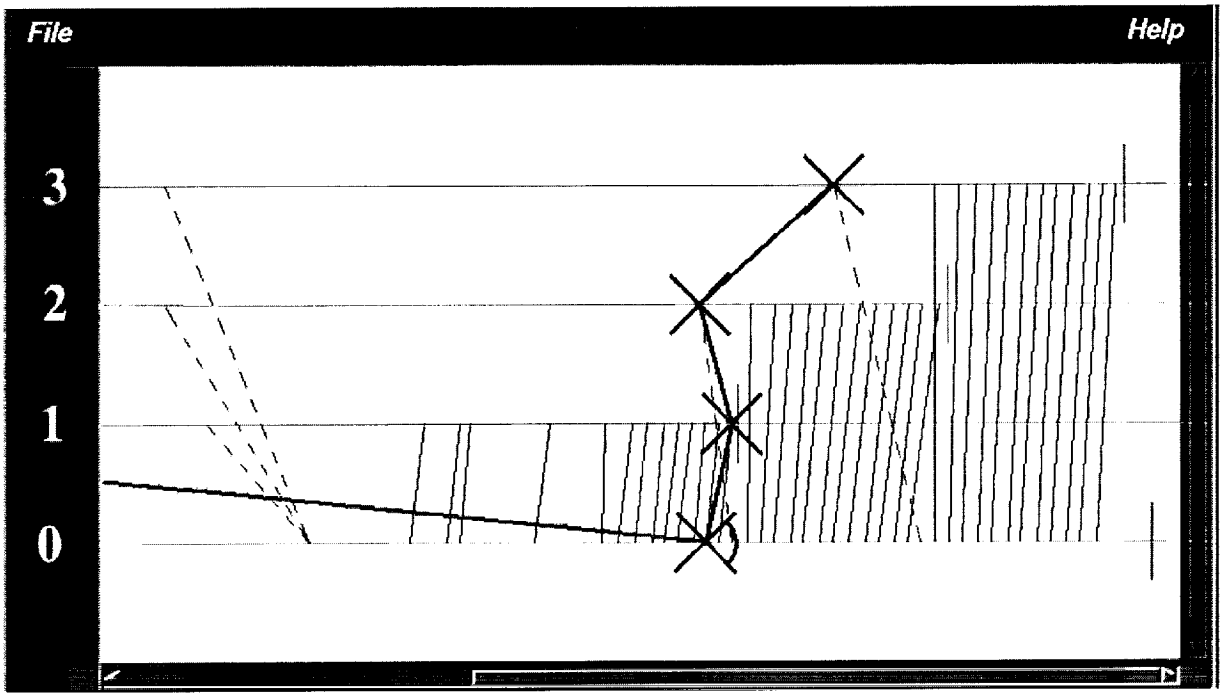

Figure 2: History displayed by p2d2 after pausing processes. Short vertical lines indicate where the processes stopped. The user has chosen a concurrency point just after the second-to-last send from process 0 to process 1 (denoted by the arc on the right-hand side of the send event).

The user opens *p2d2* and presses the RUN button, and, once it is apparent that the processes are stuck in a running state, presses the PAUSE button to force *p2d2* to update the trace display. The trace display shows that the processes never get past the distribution phase (see Figure 2), and that for some reason the number of rows of matrix *A* being sent to each process, one row per message, is smaller than expected.

See Figure 2 as well for the user-chosen concurrency point, which is just after the second-to-last send operation from process 0 to process 1. The user sets a stopline breakpoint at the future frontier of the concurrency point. After re-running, the execution stops just before the last send operation from process 0 to process 1. At that point the trace display verifies this (Figure 3), and the user is able to discover in the main *p2d2* window (Figure 4) that

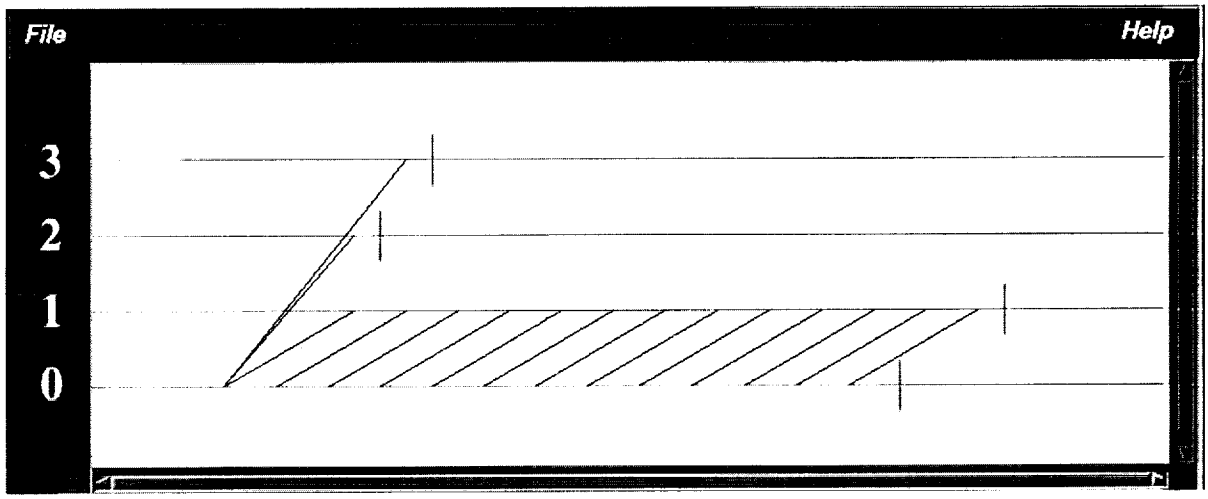

Figure 3: History displayed after replay of execution. Process 0 has stopped just before its last send to process 1, and likewise process 1 has stopped just before its last receive. Processes 2 and 3 have stopped just before receiving messages from process 0.

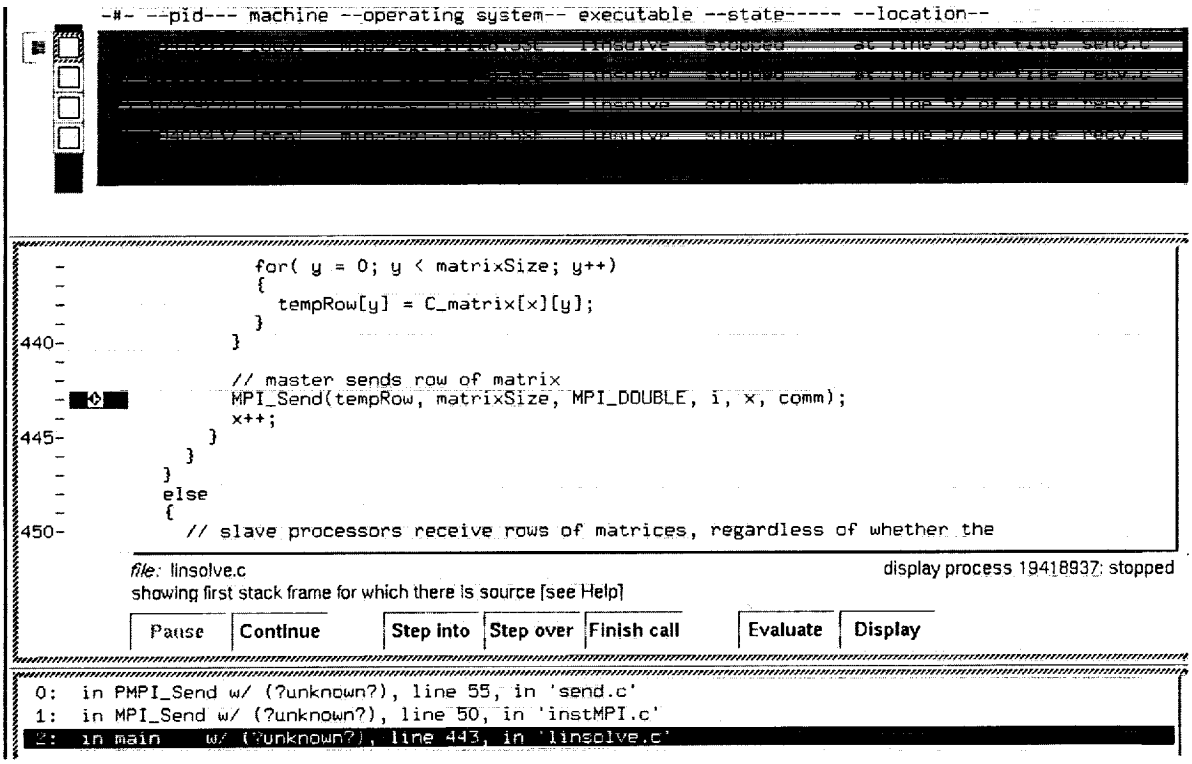

Figure 4: Stopping location of process 0 in the source code display of *p2d2*. The loop count is evaluated here to discover the programming error.

process 0 only sends each process half as many rows as it should—it reaches the end of its send loop too quickly—while the receiving processes perform the correct number of receives. This discrepancy between sends and receives results in all processes hanging. Adjusting the parameters of the send loop of process 0 fixes the problem.

5 Implementation Experiences

In order to test our hypotheses about the suitability of the compressible trace for debugging and about the degree to which it can be compressed, we implemented a trace collection mechanism to go with the trace viewer described in section 3.

We implemented trace collection for MPI programs by writing a set of wrappers using MPI's profiling interface. In effect, each wrapper routine does three things:

- it collects some preliminary trace information and, for those MPI routines that send or receive messages, creates a trace record to mark the beginning of that send or receive;
- it calls the MPI routine it is wrapping; and
- it produces a trace record describing the completion of the MPI routine.

Each record is counted during the computation. This count is used as the timeline location marker. Each time the counter is incremented during a replay, it is checked against a threshold value to see if execution should be stopped.

We implemented the viewer in *p2d2*, the distributed debugger mentioned earlier. The viewer extracts trace data from each process when it is interrupted, and then reconstructs the additional execution history represented by that data and displays it.

If the user creates a stopline in the trace, the viewer calculates the record number for each process, using the frontier calculation described in section 3.4. If execution is restarted, the debugger writes those threshold values in the newly started processes so that they will stop themselves at the appropriate time.

After implementing the wrapper routines and the trace viewer, we tested our prototype on some MPI programs.

5.1 Issues discovered during implementation

The timestamps discussed in section 2.2 should reflect the amount of CPU time used by the processes being debugged (including time spent waiting). In a debugging scenario the options for obtaining this time are limited by the constraint that time spent at a breakpoint, or other such debugging interruption, should *not* be counted as execution time. One way to avoid this is to use system calls that report execution time directly, typically as a number of clock ticks.

Our implementation on an SGI Origin 2000, running IRIX 6.5, depended on the system call *times()*, which returned clock ticks equal to 10 milliseconds of CPU time. This large clock tick size led to problems in the display of the execution history, because actual CPU time for typical MPI routines is on the order of microseconds. The logarithmic timestamp approximation thus reported a time delta of at least 10 milliseconds for each trace record, causing the reconstructed approximate absolute time to become further and further ahead of actual absolute time. This effect defeats the self-correcting property of the logarithmic time deltas, and causes the display to show messages as though they were traveling backwards in time. The operations available for interacting with the trace display are not affected, but the display can be confusing for users.

In the absence of a system call reporting actual execution time at a sufficiently fine granularity level we resorted to calculations of wall clock time that were corrected for time spent in states other than execution. Two different methods to achieve this were implemented:

- the *gdb* processes track wall clock time only when *p2d2* instructs them to do so, and
- *p2d2* tracks non-execution time in order to correct the time deltas reported by the *gdb* processes.

Both methods resulted in a trace display that had few to no messages that appeared to be traveling backwards in time, and was therefore much less confusing than the display resulting from use of the *times()* system call. However these methods have drawbacks as well, the first being a loss of accuracy caused by latency of messages sent between *p2d2* and the *gdb* processes. In the first method the wall clock time that passes while the instructive message is sent from *p2d2* to the *gdb* processes is incorrectly perceived as execution time. In the second method *p2d2* underestimates non-execution time because it marks the beginning and end of breakpoints just as it sends messages to the *gdb* processes to begin or suspend execution, not when execution is actually begun or stopped.

An additional drawback of the second method is caused by the error-correcting timestamp approximations described in section 2.2. *P2d2* tracks the amount of time spent in a non-execution state in order to subtract the amount from the time delta of the first trace record that occurs once a process begins executing again. Unfortunately the extra wall clock time spent in a non-execution state is often spread out over two or more trace records due to the error-correcting method of time delta calculation, and reconstructed timestamps for those trace records may therefore be nonsensical (negative, or close to zero). Consequently this method of wall clock correction requires that time deltas be calculated without the error-correcting attribute.

5.2 Trace sizes

One of the chief goals of our implementation was to see how small we could make the trace data and still have it be a reasonable representation of the computation. See Table 1 for sizes of trace data collected from running the NAS Parallel Benchmark [11] programs SP and LU with 4000 and 2500 iterations, respectively, on 16 processes.

The test programs were run under the Automated Instrumentation and Monitoring System (AIMS) [16], the MPI extension of the Portable Instrumented Communication Library (MPICL) [10], and *p2d2*. AIMS uses a source-to-source instrumentation method, while MPICL is an instrumented communication library like the one we have implemented. The resulting number of MPI events was about 250,000 for SP, and from 635,000 to 1.2 million for LU. All numbers describe the summation of trace data collected across all processes, averaged over 10 runs. Note that compression numbers for AIMS and MPICL were obtained by compressing with *gzip*.

Of particular interest in these numbers is the degree to which our new format can be compressed. Extrapolating from the current results indicates that 220 million MPI events could be stored in an 80 megabyte buffer in the process's address space. However, this is a low estimate because larger trace files facilitate better compression, and indeed our largest test to date confirms this with 50.8 million events in a 16.28 megabyte buffer (which scales to 220 million events in a 70.5 megabyte buffer).

5.3 Limitations in the *p2d2*+MPI implementation

The choice of MPI as the library to use in target programs leads to some limitations in the implementation of trace viewing in *p2d2*. In general, these limitations are due to MPI's broad functionality.

Missed messages: The MPI standard allows for non-blocking communication, and each message sent in this way must have an `MPI_Request` data structure allocated. These structures can be freed using the `MPI_REQUEST_FREE` call before a non-blocking message has completed its transfer, leaving the actual status of the message unknown to MPI and to our implementation. The resulting behavior in the program being debugged may indicate successful completion of the message transfer, but the trace display will not show any such message.

Trace Source	Trace Type	Size in MBytes		% Compression
		Before	After	
SP	*p2d2*	182.4	2.5	98.6
	AIMS	237.4	36.5	84.6
	MPICL	381.8	65.5	82.8
LU	*p2d2*	724.7	4.6	99.3
	AIMS	647.4	120.7	81.3
	MPICL	900.6	141.7	84.3

Table 1: Trace Size Comparison

Imperfect stopline frontiers: Ideally the frontiers described in section 3.4 will contain events such that every process involved in a debugging session can suspend itself during replay—that is, no process will block waiting for an event from a stopped process. This is achieved in MPI if all outstanding message sends and receives are completed before an attempt is made to suspend. Otherwise the process remains in a running state waiting for the send or receive to complete.

We discovered that ideal frontiers could always be found with message passing programs that utilized just one MPI communicator. Using more than one communicator can lead to situations in which the ordering of message sends is different than the ordering of message receives (across all communicators), and results in processes that are waiting for sends or receives to complete at the time of the stopline. Further work is necessary to alleviate this problem.

Imperfect rollback: In order to reverse execute just one message event in the execution history it is often necessary to leave processes in a state where a message send or receive is incomplete. As described above this leads to processes that remain in a running state, and reduces the usefulness of the rollback feature in those cases.

6 Related Work

Because the execution of message passing programs is more complex than that of serial programs, more powerful debugging techniques are required to isolate errors. The increased complexity comes from:

- interprocess communications and synchronization,
- an increased number of steps to compute the same result, and
- increased potential for nondeterministic behavior.

Several approaches for dealing with these sources of complexity have been researched.

One approach that is common to several projects provides a replay mechanism that makes it possible to use the error isolation method of serial codes. Once a deterministic execution, at least for debugging purposes, can be guaranteed, the user is free to re-execute the code in an attempt to zero in on the bug. In general, the replay mechanisms collect trace information during an initial run that records critical event ordering information. It then orchestrates re-executions so that the event orderings are honored.

The *Instant Replay* work of LeBlanc and Mellor-Crummey [6] together with follow-up work on software instruction counters [8] served as a conceptual foundation for an integrated debugging and performance analysis toolkit [7] for shared memory codes. The *Pangaea* [4] project used logging information to enforce event ordering during replay of PVM [15] programs.

There has also been work on minimizing the number of trace records collected in order to guarantee event ordering during replay. The *Optimal tracing and replay* [12][13][14] work is applicable both for shared memory and message passing codes.

One other approach of note for debugging parallel programs is the behavioral approach used by *Ariadne* [1]. It provides a post-mortem facility for comparing actual execution with a user-specified model of behavior.

An earlier paper from the *p2d2* project described the full integration of trace visualization and a replay mechanism in a state-based debugger for message passing programs [3]. It included features for consistent trace-based stoplines and for a replay-based implementation of UNDO. That work, however, did not solve the problems inherent in the real-time display of the trace. It also made no attempt to minimize the amount of trace data collected.

In the area of minimizing trace information, Yan and Schmidt experimented with fixed-size trace files [17]. Their work is based on the observation that program behavior follows patterns. The trace collection routines attempt to represent repeated patterns during execution with formulae containing repetition counts. The number of time stamps in the trace is greatly reduced and as a result, the trace that is reconstructed from the formulae has interpolated time stamps that may not be accurate. Furthermore, the formula-producing mechanism in the trace collection has a limited collection of building blocks from which to build patterns.

7 Conclusions & Future Work

The ability to see and interact with an abstract view of the execution history of a message-passing program can provide a significant benefit to debugging. For example, a facility for displaying an anomalous message-passing pattern and then stopping before it occurs during a re-execution can simplify the job of isolating the bug that causes it.

In this work we have described the implementation of a practical mechanism for collecting and displaying trace information during a debugging session. Our approach solves the problems of displaying up-to-date trace information. It also introduces a trace format that is highly compressible while still providing information adequate for debugging purposes. We make the mechanism convenient for users to access by incorporating the trace collection in a set of wrappers for the MPI communication library.

There are several debugger operations that use the trace display: consistent stoplines, undo, and rollback. They are all implemented using a controlled replay mechanism that executes without debugger interpretation in target processes until the appropriate position in the timeline is reached. The replay technique permits

execution at virtually the same speed as uninstrumented code.

In the future, we want to experiment with different approximation schemes for the time stamp deltas that appear in trace records. We would also like to see how additional information, such as program source location, can be added to the trace records without sacrificing compressibility. We will also experiment with different skewing techniques in an attempt to minimize the number of messages that appear to go backwards in time.

Acknowledgments

This paper builds on a long tradition of trace visualization work by our late colleague Lou Lopez. At the time of his passing in 1999 he was beginning the experimental work on compressible traces that is contained here.

We also want to express appreciation to Rob Van der Wijngaart and Michael Frumkin for helpful suggestions on improving the paper. Rob also helped us with our characterization of communication patterns in CFD applications.

References

[1] J. Cuny, C. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, D. Stemple. The Ariadne Debugger: Scalable Application of Event-Based Abstraction. *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993, San Diego, CA, pp.85-95.

[2] The *Dyninst* API. http://www.cs.umd.edu/projects/dyninstAPI.

[3] M. Frumkin, R. Hood, and L. Lopez. Trace-driven Debugging of Message Passing Programs. *Proceedings of IPPS'98*, Orlando, FL, 1998.

[4] L. Hicks, F. Berman. Debugging Heterogeneous Applications with Pangaea. *Proceedings of SPDT'96: SIGMETRICS. Symposium on Parallel and Distributed Tools.* May 1996. Philadelphia, PA, pp. 41-50.

[5] R. Hood. The *p2d2* Project: Building a Portable Distributed Debugger. *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996, Philadelphia, PA, pp. 126-137.

[6] T. J. LeBlanc, J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. on Computers C-36* (4), April 1987, pp. 471-482.

[7] T. J. LeBlanc, J. M. Mellor-Crummey, R.J. Fowler. Analyzing Parallel Programs Execution Using Multiple Views. *Journal of Parallel and Distr. Comp.* 9 (2), pp. 203-217 (1990).

[8] J. M. Mellor-Crummey, T. J. LeBlanc. A Software Instruction Counter. *Third International Conference on Architectural Support for Programming Languages and Operating Systems.* April 1989, pp. 78-86.

[9] MPI: A Message-Passing Interface Standard. June 1995.

[10] The MPICL Portable Instrumentation Library. http://www.csm.ornl.gov/picl.

[11] The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB.

[12] R.H.B. Netzer, T. W. Brennan, S.K. Damodaran-Kamal. Debugging Race Conditions in Message Passing Programs. *Proceedings of SPDT'96 SIGMETRICS. Symposium on Parallel and Distributed Tools.* May 1996, Philadelphia, PA, pp. 31-40.

[13] R.H.B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993, San-Diego, CA, pp. 1-11.

[14] R.H.B. Netzer, B.P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs *Supercomputing 92*, Minneapolis, MN.

[15] V. Sunderam. PVM. A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience* 2(4), pp. 315-339 (1990).

[16] J. Yan, S. Sarukkai and P. Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software—Practice and Experience* 25 (4), pp. 429-461 (1995).

[17] J. Yan and M. Schmidt. Constructing Space-Time Views from Fixed Size Trace Files—Getting the Best of Both Worlds. *Proceedings of Parallel Computing '97 (ParCo97)*, Bonn, 1997.

[18] The *Zlib* compression library. http://www.info-zip.org/pub/infozip/zlib.