Trace-Driven Debugging of Message Passing Programs*

Michael Frumkin, Robert Hood, Louis Lopez
MRJ Technology Solutions—NAS Systems Division
NASA Ames Research Center—M/S T27A-1
Moffett Field, CA 94035
{frumkin,rhood,llopez}@nas.nasa.gov

Abstract

In this paper we report on features added to a parallel debugger to simplify the debugging of message passing programs. These features include replay, setting consistent breakpoints based on interprocess event causality, a parallel undo operation, and communication supervision. These features all use trace information collected during the execution of the program being debugged. We used a number of different instrumentation techniques to collect traces. We also implemented trace displays using two different trace visualization systems. The implementation was tested on an SGI Power Challenge cluster and a network of SGI workstations.

1 Introduction

Debugging of parallel programs can be significantly facilitated by providing the user with both a big picture of what has happened during program execution and a way of steering the debugging session based on this picture. This steering could include a replay mechanism that supports a breakpoint across all processes that is consistent with an event of the user's choosing from the program execution history. It could also include an *undo* operation and the monitoring of sends and receives.

A number of papers have realized the significance of a big picture for debugging parallel programs and there are a few parallel debuggers that provide subsets of the capabilities mentioned above [4][6][8][9][10]. A survey by Kraemer and Stasko [6] describes earlier approaches.

In this paper we present a clear, consistent design and implementation of these features that is based on the collection and visualization of program traces and on a debugger managed replay mechanism. Our approach reduces debugging effort to a level comparable with program monitoring and is applicable to single-threaded message passing programs written in Fortran or C.

*Supported through NASA contract NAS 2-14303.

We have implemented these trace-driven debugging features in *p2d2*, a portable distributed debugger developed at NASA Ames [5] [`http://www.nas.nasa.gov/Tools/p2d2`]. The debugger was designed to handle computationally intensive programs that are distributed across a large number of processors. As such it supports debugging of PVM [19] and MPI [13][14] programs on a variety of platforms including the IBM SP, SGI Power Challenge, and networks of Unix workstations.

In the remainder of this paper we detail our strategy for providing trace-driven debugging. In the next section we discuss three methods for acquiring trace information which differ in the granularity of the information provided and the effort required on the part of the user. In section 3 we describe three methods for visualizing the trace data in *p2d2*. Section 4 discusses execution steering through the use of breakpoint stoplines in the time-space diagram of the target program execution. In section 5 we compare our approach with previous execution replay techniques.

2 History acquisition and controlled replay

A debugger that provides operations based on the program execution history needs an efficient way of acquiring that information. In its most general form, the execution history includes a sequence of program states for every process in the computation, where a single sequence would have one program state for every statement executed so far. Since it is clearly prohibitive to collect that much information, we settle for a smaller representation of the execution history that is based on a trace of significant events that take place during execution.

We use three different approaches for acquiring the trace data. These approaches offer distinct levels of user convenience, history detail, and execution overhead. The first method is built on top of *AIMS* [20], which is a toolkit designed for the post-mortem analysis of execution. It uses source-to-source transformations to insert calls to functions in a monitoring library. The second method utilizes a newly developed tool, `uinst`, to insert calls to a user-level monitoring function into the assembler code of the program. This instrumentation plays a key role in execution replay and the *undo* operation. The third method for acquiring trace data uses instrumented wrappers for com-

munication library routines. The PMPI_ profiling interface of MPI provides a particularly convenient way of instrumenting calls to that library.

The trace-driven debugging features we want are implemented with a controlled replay mechanism. The key idea is to put tags in the execution trace that allow mapping from a particular trace record to the point of its generation. We call such a tag an *execution marker*. The instrumentation software is responsible for generating the markers and putting them in the trace data. In addition, during a replay the instrumentation routines need to test every marker at the time of generation to see if it is one of interest to the debugger.

2.1 Source-to-source instrumentation

Instrumentation at the source code level allows execution history acquisition with an arbitrary level of resolution ranging from function entry/exit to individual assignment statements. The instrumentation can be done manually or through the use of instrumentation tools. At execution time an instrumented program generates a trace file which can be interpreted and displayed by the debugger.

We used *AIMS*, the Automated Instrumentation and Monitoring System [20] [http://www.nas.nasa.gov/Tools/AIMS], to instrument the code and collect trace data during execution. *AIMS* consists of a suite of software tools for measuring and analyzing the performance of FORTRAN and C message passing programs. It allows selective insertion of calls to performance monitoring routines into the source program. It then collects the program trace data produced at run time and allows the user to examine it with visualization tools.

In order to use an *AIMS* trace in the debugger, we had to address two problems. First, *AIMS* is a tool designed for *post-mortem* analysis and *p2d2* needs trace data *during* execution. This was handled by adding a monitor function that flushes trace information on demand. The second problem was to extend the *AIMS* monitor functions with a controlled replay mechanism by having them generate and test the execution markers described above.

2.2 Compiler-inserted instrumentation

The flexibility of the source level instrumentation strategy discussed in the previous section has a cost: it requires substantial user awareness of the instrumentation process. At the least the user must modify makefiles to add a step to the compilation process. The user must also cope with the existence of the set of transformed source files. In fact, if the debugger is not provided with a description of how files were instrumented, the user may be required to debug the transformed code rather than the original code.

At the expense of flexible instrumentation levels, we can shift most of the user burden of instrumentation to a compiler. For our purposes it is sufficient to have an instrumentation call put at the end of the prologue code of every user function. We suggest that this could be done automatically in code compiled with the debugging flag "-g".

Since no compiler does this yet, we tested this approach a little differently. We took advantage of the fact that some compilers, such as the Free Software Foundation compiler, *gcc*, put a function call to mcount in the prologue of any function compiled with the "-p" flag. We replaced the assembly language call to mcount with a call to our own function UserMonitor. The UserMonitor function performs all the monitoring and tracing actions we need.

We also wrote a tool, uinst, which inserts calls to UserMonitor. The main problem in changing from an mcount call to a UserMonitor one is that the provided mcount function is written in assembler and the call to it does not follow the standard calling convention. To get around this, we changed the call to mcount to be a call to ucount, and then call UserMonitor from ucount. Because it does not follow the calling convention, the ucount function was written in assembler code.

In our prototype system, instrumentation is inserted automatically by changing the compilation command

```
gcc -c -g file.c
```

to

```
gcc -p -g -S file.c
uinst file.s
gcc -c file.s
```

where the unist program scans through the assembler code and replaces calls to mcount with calls to ucount. In addition, the linking step needs to include ".o" files for the ucount and UserMonitor functions. In the future, we will modify *gcc* so that it inserts the necessary instrumentation directly when invoked with "-g". In this way, the user compilation commands will be completely unchanged.

The UserMonitor function is well-positioned both to generate the history information needed by the debugger and to test during a replay to see if the marker being generated is one the debugger is interested in. In its current implementation, the function increments a single global counter, records the address it was called from together with the first two arguments passed to it, and tests to see if the global counter has reached a threshold value which can be set by the debugger.

Since UserMonitor could be called by every function in the program, we should be concerned about the overhead incurred by its use. Table 1 shows that the overhead imposed by the UserMonitor function is quite small in the typical case (an implementation of Strassen's matrix multiplication algorithm) and is reasonable in the worst case (a recursive Fibonacci function [11]).

2.3 Wrappers on message passing library functions

One drawback of the instrumentation strategy in Section 2.2 is the machine-specific nature of the modification to *gcc* (or alternatively the implementation of the

	Strassen matrix multiply (4 procs)		Fibonacci	
Input size/value	96·128·112	192·256·224	34	35
Number of calls	136	136	18454930	29860704
Time (uninstr.)	8.19	28.72	5.17	8.36
Time (instr.)	8.46	28.77	20.98	34.12

TABLE 1. Instrumentation overhead (seconds).

`uinst` function). By reducing the granularity of the history generation we can provide a highly portable trace collection mechanism. This instrumentation approach wraps calls to the communication library with instrumentation collection. In addition to generating the execution marker needed for controlled replay, this instrumentation technique can also be used to create a communication graph [6] and to detect race conditions [15] in the message passing.

The MPI standard provides a particularly convenient way of accomplishing this. The standard mandates a profiling interface [13,p.201] that essentially makes every library function available under two distinct names, `MPI_`*name* and `PMPI_`*name*. For example, the `MPI_Init` function can also be called by invoking `PMPI_Init`.

Using this technique we supply an instrumented MPI library that acts as a front-end to the `PMPI_` functions. For example, we supply an `MPI_Send` that generates history information and then calls `PMPI_Send`. When the user links with the debugging version of the MPI library, the history collection is automatic. We envision that MPI-knowledgeable compile commands, such as `mrf77` (a version of `f77` used to compile MPI programs at NAS), could perform the necessary link automatically when invoked with the "`-g`" flag.

For our prototyping work, we wrote our own versions of `MPI_Send` and `MPI_Receive`. These two functions contain a call to a dummy function, passing in the sender/receiver rank and the message tag, followed by a call to the `PMPI_` version. Then the `MPI_Send`, `MPI_Receive`, and the dummy function were instrumented with `uinst`. By instrumenting the three functions, the `UserMonitor` function can log the event, including the function making the request and the message endpoints and tag. This information can then be used to build a communication graph and tie it to the parts of the program source responsible for the events.

This interface makes instrumentation and collecting data on communications very simple and efficient. If the low resolution of events is a problem, the techniques of the previous two sections can be used in combination with this one.

3 Displaying history information in *p2d2*

In the previous section we described three instrumentation methods for collecting the execution history. In this section we describe how we extended *p2d2* to make it history aware and give it the ability to display an execution history. Our goals for the display are to have it provide an abstract picture of what has happened so far during execution and to provide a navigation mechanism for identifying points of interest in the execution.

In our current implementation, history is represented by an *AIMS* trace file [20]. The trace contains a record for each execution of each instrumented program construct, such as a communication event. A record identifies the construct by giving its program location, the *id* of the process that executed the construct, and the start and end time of the construct execution. In addition, if the construct is a message passing operation, the record contains the message tag together with the source and destination of the message. The size of trace file can be controlled by selectively instrumenting constructs and by toggling the collection on and off in the monitor. The general scheme of interaction between *p2d2* and the visualization tools is shown in Figure 1.

3.1 Time-space diagrams

We used two trace visualizers to display history: *VK*, the visualizer from AIMS, and the *NTV*, the NAS Trace Visualizer [`http://www.nas.nasa.gov/Tools/ntv`]. Both display execution history as a time-space diagram. Each construct is represented by a bar positioned according to its process number and start/end times. The bar is colored depending on the type of the construct. Each message is represented by a straight line segment connecting *(time_sent, source)* and *(time_received, destination)* points of the time-space display (see Figures 2 and 3).

Both visualizers provide a way to relate constructs back to the source program. For example, in either tool, clicking on a bar representing a message line can identify the location of the send or receive in the source code.

The two visualizers differ in their display paradigms. *NTV* provides the user with the entire trace file at one time and allows selective zooming and panning to find events of

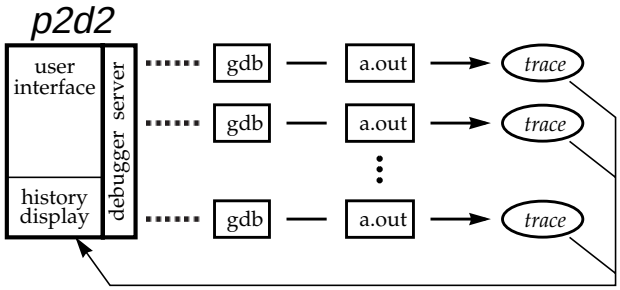

FIGURE 1. History visualization in *p2d2*.

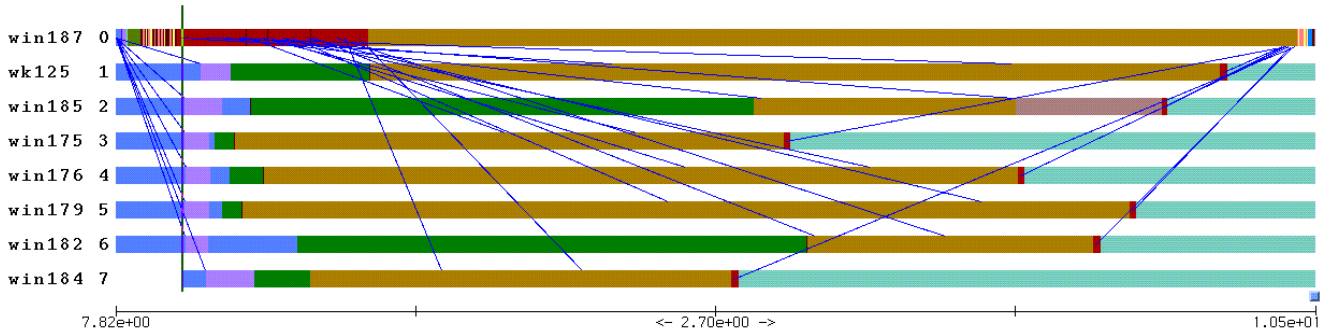

FIGURE 2. History displayed with NTV. Angled lines represent messages; the vertical line near the left side represents the stopline (see text).

interest. *VK*, on the other hand, gives the user a window into the trace file and provides an animated view of the events of execution. The user can scroll through the history in both directions and change the time scale.

In order to add the trace-driven debugging features to *p2d2*, we integrated both visualizers into the debugger. In the case of *NTV*, this amounted to linking in the Ben trace visualization library upon which *NTV* is built. The Ben interface allows the debugger to find out what the execution markers are at the point of a mouse click in the time line. It also provides an indicator (a vertical line) that the debugger can use to mark a point in the history. This combination of features permits *p2d2* to implement a *stopline*, that is, a breakpoint in the timeline. When the user requests one at a particular point, the debugger can find out the corresponding execution markers for each of the processes and indicate the position of the breakpoint in the timeline. When execution is replayed, the execution markers tell the debugger when to stop each of the processes.

In the case of *VK*, we have *p2d2* start up the visualizer as a separate process. While communicating across the address spaces is somewhat clumsier than in the Ben-*p2d2* case, analyzing the trace data is easier because it all takes place outside of *p2d2*. Using *VK*, for example, we can easily show the causal past (or future) of a selected event, so that the user can skip events that do not affect (or are not affected by) the current event. While this feature was also implemented with the *NTV* visualizer, it required more processing of trace data within *p2d2*.

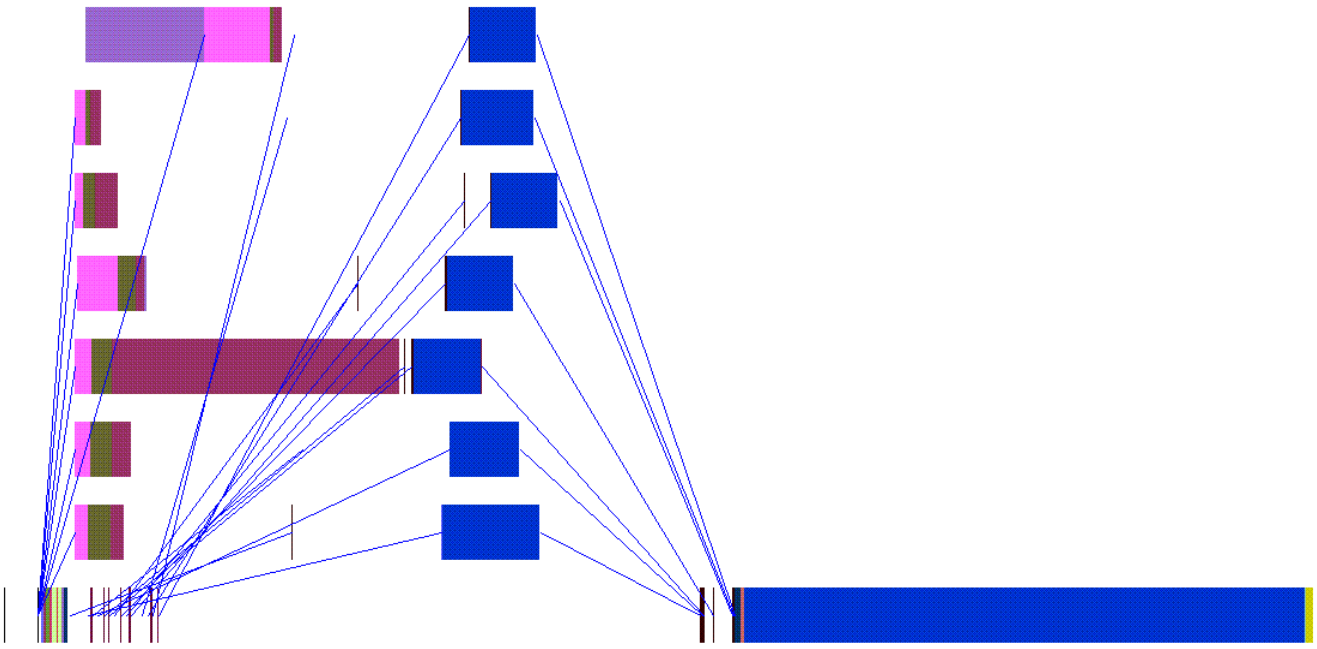

FIGURE 3. History displayed with *VK*. A trace of Strassen's matrix multiplication running on 8 processes. Process 0 (at the bottom) distributes pairs of submatrices among the other processes (each send is shown as a separate message). Then process 0 receives 7 partial results and combines them into the final result.

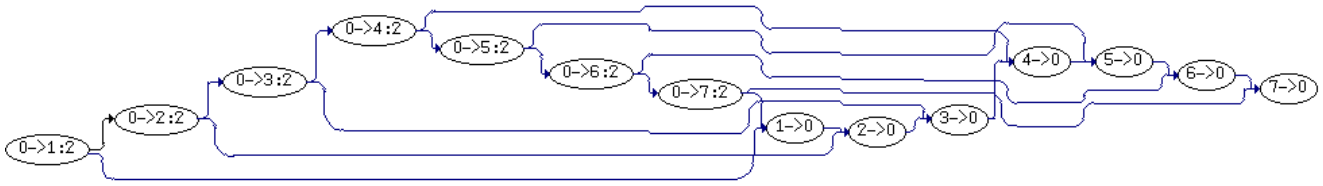

FIGURE 4. Communication graph of Strassen's algorithm implementation. Each node corresponds to one or two messages. The arcs describe causality of messages.

3.2 Dynamic call and communication graphs

In order to support the trace analysis necessary for features such as indicating the causal past or future of an event, we implemented a graph abstraction of the execution history. It captures event causality while skipping less important information such as start time and end time of execution of a particular construct. This abstraction provides a coarser view of history while preserving the event causality relation.

The *trace graph* of the execution is a graph whose vertex set consists of a node for each function in the program and a node for each communication channel (one channel per pair of processes). There are two types of arcs in the trace graph. Each function call is represented with an arc from the node of the caller to the callee node. Each message send/receive is represented with an arc from the function performing the send/receive to the channel involved. The message "non-overtaking" property specified in the MPI standard [13, p.30] allows a unique matching of send arcs with receive arcs incident to the same channel and having the same message tag. (Recall that we are dealing with single-threaded applications).

Projection of the trace graph onto a particular process (that is removing all nodes belonging to other processes and channels and their incident arcs) gives us a dynamic call graph [3][6][8] of the process. A simple transformation of the trace graph gives us a communication graph [6] (see Figure 4).With these abstractions we can provide tools for helping analyze the anomalies in message traffic. For example, we can isolate the unmatched send/receives in the execution and detect some types of deadlocks. In Section 4.4 we give an example of how this can be done.

4 Trace-driven debugging

In a conventional debugging session the user must make decisions regarding the granularity of computation to be performed. It is all too easy to spend ten or fifteen minutes getting near a bug and then accidently issue a "*step over*" command instead of "*step into*". What could happen then, of course, is a program crash caused by some statement in the function stepped over, with no additional information about the real bug. In this section we show how execution history information can smooth the debugging process by providing *replay* and *undo* capabilities.

4.1 Trace-driven replay

The ability to visualize the program's execution gives the user a powerful debugging tool. In a situation where a program crashes and a post-mortem debugging session sheds no light on the bug, the user can instrument the program and get an execution trace to the point of the crash. At that point a simple display of the trace can give the user a hint as to where the execution goes wrong. By setting a stopline and replaying, the user can have the execution stop before the problem occurs and use standard debugging techniques to locate the problem.

To set a stopline, the user identifies a particular event in the timeline and then invokes the "*set stopline*" operation. The meaning of the stopline is that execution should stop at that point in the process where the event was selected. Other processes will be stopped at a point consistent with that point, *i.e.*, at some point guaranteed to be concurrent with the selected point.

In Figure 5 we show an example trace from a non-working distributed implementation of Strassen matrix multiplication. Apparently, processes 0 and 7 are failing to make progress. By increasing the magnification (see Figure 6) the user can examine the message bundle and notice that process 7 is not behaving like processes 1-6. In

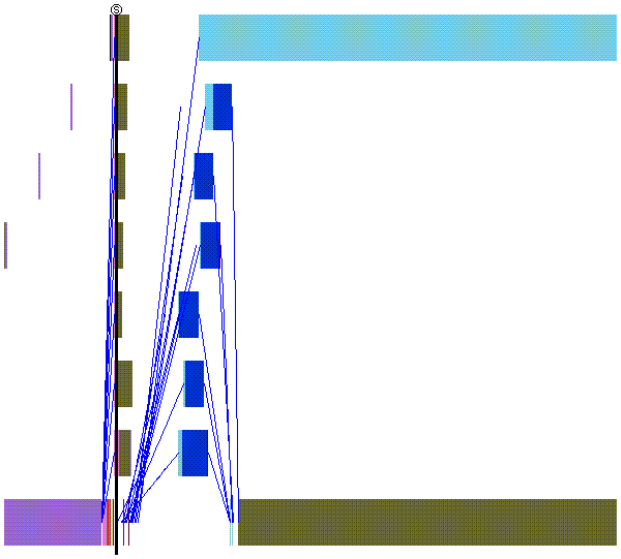

FIGURE 5. Process 0 (at the bottom) and process 7 (at the top) are blocked in receives waiting for data from each other.

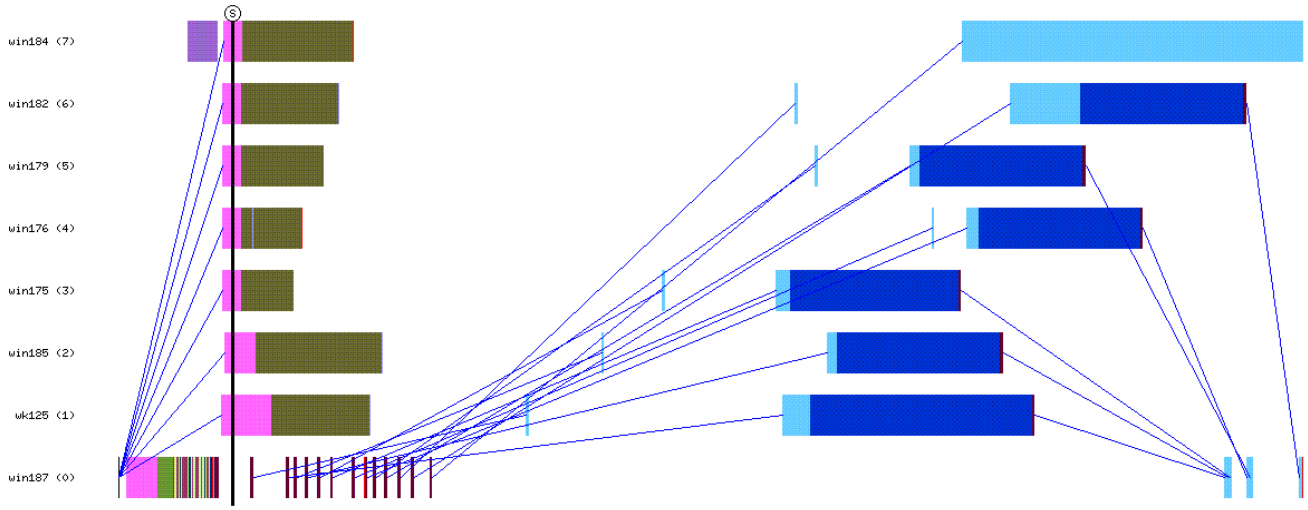

FIGURE 6. Missed message from process 0 to process 7. The correct message sequence is shown in Figure 3. The vertical stopline (on the left side) gives a consistent set of breakpoints for replay.

particular, processes 1-6 each have a small vertical tick before a longer computation bar. Process 7 is missing that tick. Closer examination reveals that processes 1-6 each receive 2 messages and process 7 only receives 1. (The tick mark is actually a very short period of computation at the time of the first receive.) At this point the user can set a stopline somewhere before the first send in the group. The stopline will be communicated to *p2d2* as a set of breakpoints along with the execution markers indicating the corresponding states in the execution. This information is sufficient for *p2d2* to perform a replay. (See the discussion on nondeterminism control in Section 6)

When the user requests a re-execution, the debugger restarts the computation, and as part of that, stores the execution markers in the `UserMonitor` threshold variables discussed in Section 2.2. When the routine generates an execution marker equal to the threshold value, it triggers a debugger-set breakpoint. When this occurs in our example, a few "*step*" operations would lead the user to the loop of `MatrSend` (see Figure 7). Stepping through the loop, the user will find that `jres` should be replaced by `jres+1` in line 161.

When implementing a stopline, it is important for the debugger to use a consistent set of breakpoints [18]. The consistency of breakpoints derived from the stopline follows from the causality of communications in the trace file, *i.e.*, no message was received before it was sent. In other words, the stopline passes through a concurrent set of events. The actual concurrency region of the event can be significantly wider than the set of events the stopline passes through. The concurrency area lies between an event's past and future. The past of the event is defined as the set of events that are guaranteed to have happened before it; an event is in the future of the current event if the former happened before the latter.

In order to depict the past and future of an event we use the notion of *consistent frontier* [15]. It is defined as a set of events in which no event happens before another. Lack of circular message dependencies in the trace file guarantees that set of most recent events in the past is a consistent frontier (past frontier). The same is true for the set of earliest events of the future (future frontier). We can draw past- and future- frontiers based on simple analysis of the communication graph.

In Figure 8 we shown an example of past- and future-frontiers in a trace of the NAS Parallel Benchmark LU. In the example, the user clicked at the point indicated by the circle. The debugger then calculated the past- and future-frontiers and displayed them in the timeline. While not currently implemented, these frontiers could be used for setting a stopline. That is, the user could be given a choice of stopping execution in each process either immediately after the point where it could last affect the selected state or immediately before the point where it could first be affected by the selected state. This would amount to implementing the stopline either along the past- or future- frontier, rather than as a vertical slice through the history.

4.2 *Undo* of computation

The presence of execution markers in the computation allows us to implement, via replay, an *undo* operation that has the effect of returning the process states to a point very near their location before the most recent resumption operation (*e.g.*, "*step*" or "*continue*"). Every time a target process stops, *p2d2* records its execution marker. If an *undo* operation is requested, the debugger replays the program, setting the threshold variables of `UserMonitor`. That routine will then ensure that each process execution stops at the last creation of an execution tag preceding the desired state.

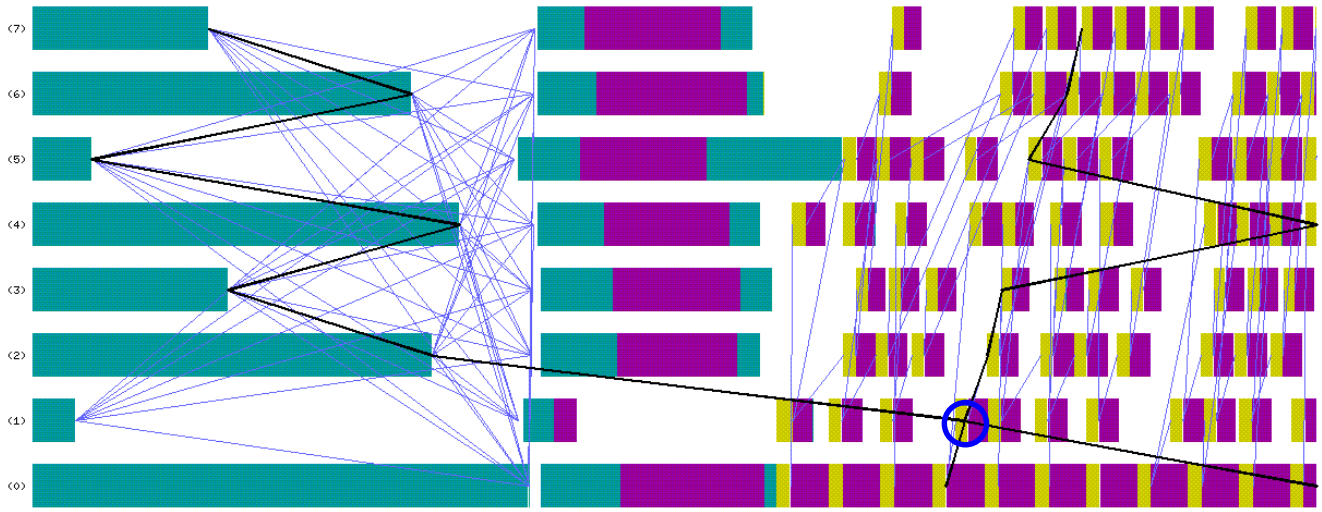

FIGURE 8. Past and future frontiers of a time point in a specific processor. The user selected the point indicated by the circle. The timeline display then calculated the region of the computation that is concurrent with that point. The concurrency region is shown between the slanted black lines.

In a replay, the behavior of nondeterministic statements (such as statements using the `MPI_ANY_SOURCE` wild card) can be controlled by *p2d2* with the information available in the program trace. This ensures that the replay has identical event causality with the original program execution.

4.3 Fast navigation of history

An execution history can be huge and often won't fit into memory, so an additional data structure is required for fast navigation through it. *P2d2* represents a history by a trace graph which is built as the execution is running. A brief description of the trace graph is given in section 3.3. The use of a trace graph gives us a fast way of locating an event in history and displaying it the time-space diagram.

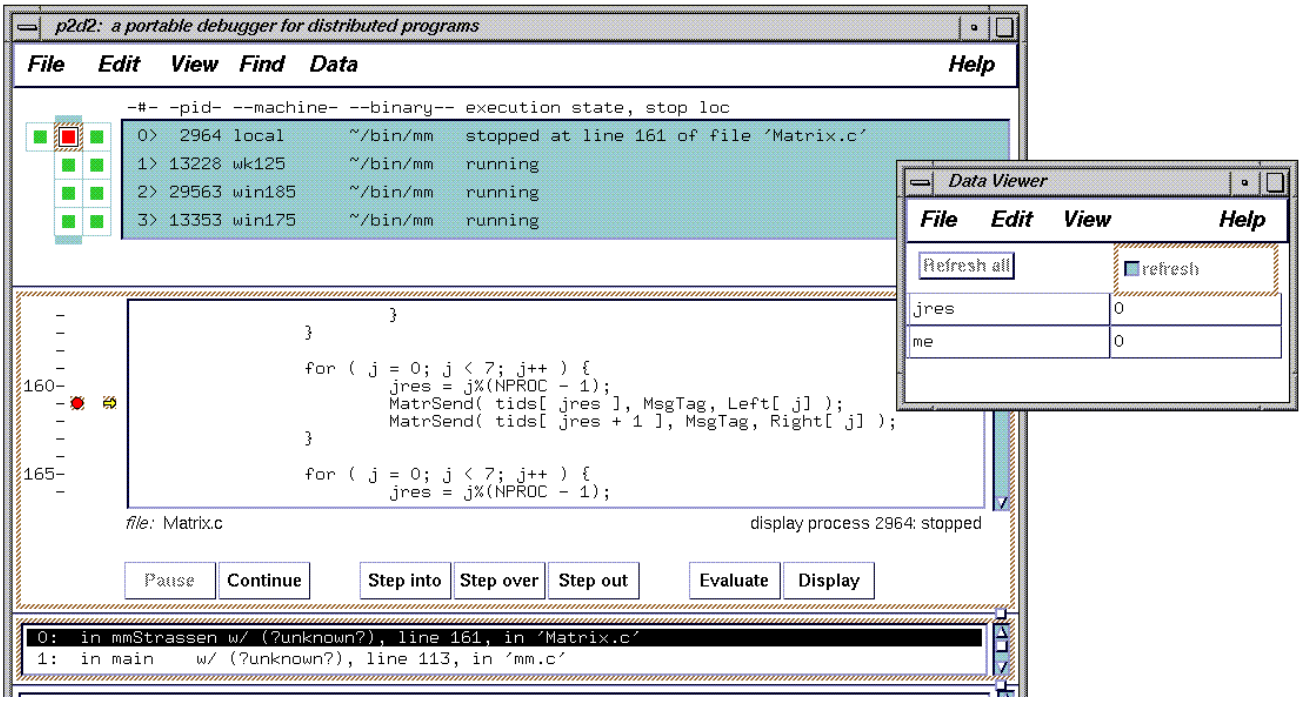

FIGURE 7. Identification of the incorrect send destination with *p2d2.*

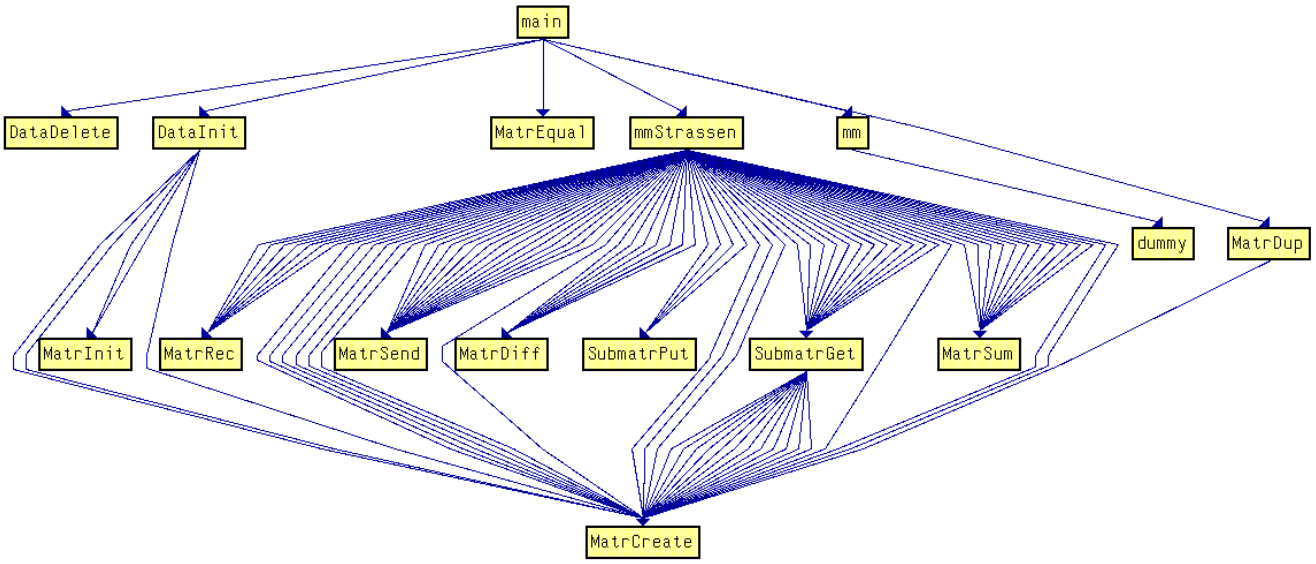

FIGURE 9. Dynamic call graph from Strassen example. Multiple arcs show multiple function calls. The number of calls per arc is adjustable. Each arc has an image in the execution trace. The graph was converted to VCG format displayed with the *xvcg* graph layout tool (`http://www.cs.uni-sb.de/~sander/gspapers.html`).

The number of nodes of the trace graph is bounded by the number of program functions times the number of processors plus the square of the number of processors. The number of arcs in the graph is the sum of the number of calls made by all processes plus twice the number of messages. In order to keep the number of arcs in the trace graph independent of the execution length, we use the dissemination technique: if the number of arcs incident to a node exceeds a limit, we merge every other arc with the previous one. This technique allows us to control the size of the history at the cost of some resolution. If the user wants to zoom in on a particular event, the required arcs are reconstructed by rescanning the appropriate portion of the trace file.

In addition to a trace graph, *p2d2* uses a call graph [3][6][8] and a communication graph [6]. These graphs are extracted by *p2d2* from the trace graph. Currently the user can display them either in text or in graphical form. Figure 9 shows an example of the call graph display for the Strassen matrix multiply example.

4.4 History analysis

The execution history can be analyzed in order to make it easier for the user to comprehend the execution. The first level of analysis is done at the level of the call graph. For every function, the calls made while the function is active are classified into actions and the call graph is transformed into actions graph. The action graph represents history with less resolution than the time-space diagram and makes it more understandable.

The debugger maintains a list of unmatched sends and receives. The list is updated as execution progresses. When a send or receive is matched, the pair is added as a node in the communication graph. As soon as the communication graph has been built, the user is informed about the unmatched send/receives. At this point, information about intertwined messages [13,p.31] is also available to the user. If the user's program is single-threaded, there are no interruptions, and the wildcard `MPI_ANY_SOURCE` is not used, then the MPI standard guarantees that the message passing code is deterministic [13][14]. If however the program is multithreaded, then message racing can occur. In this case the user might want to turn on the race detection feature of the debugger. When provided with the history trace, the debugger is also able to detect deadlocks due to circular dependency in sends or receives.

5 Related work

The debugging of parallel message passing programs is more complex than that of conventional programs. The increased complexity comes from at least three sources: process interaction through communication, more events in a parallel program than in a sequential program performing the same task, and more sources of nondeterminism in a parallel program. In order to cope with the added complexity, a number of techniques and tools have been developed.

Many of these tools are based on the collection of data during program execution and controlled replay of the program using the collected data. *Instant replay* [7] gives a conceptual solution to the problem of re-execution of parallel program. (For earlier work, see the on-line bibliography by Pancake and Netzer [`http://www.cs.orst.edu/~pancake/papers/biblio.html`].) It creates an event his-

tory for each process of the program and uses the history to steer the replay.

This technique was later enhanced through the use of an efficient history recording technique called *software instruction count (SIC)* [11]. The *SIC* technique counts the branches of a program which, together with the program counter, allows to label individual program states. It was used for replaying parallel programs and for organizing watchpoints. The technique takes advantage of a flag to the *gnu* compiler that requests a particular register not be used. The counting is then done with assembly-level instrumentation.

The instant replay method was used in an integrated toolkit for debugging and performance analysis [8] of shared memory parallel programs. This toolkit supports program trace collection and trace visualization. It has a programmable interface for user queries. Event resolution in this toolkit was limited to the accessing of shared variables, process communication events, and synchronization events. Display and analysis of the execution history is based on the graph abstraction of traces.

Ariadne and *Breezy* [2][18] are event- and state-based debugging facilities for parallel programs in the *TAU* programming environment. Ariadne provides a facility for postmortem analysis of the execution of parallel programs and is able to match a user-specified model with the actual behavior captured in event traces. It uses an execution history graph. Breezy supports the setting of consistent breakpoints.

Pangaea [4] uses instrumentation of PVM programs for event logging. The event log is used to enforce the same event ordering in each execution. *Pangaea* supports replay as well as postmortem analysis. *Panorama* [10] creates a variety of graphical views of the parallel program and communicates with an existing multicomputer debugger.

The *Optimal tracing and replay* [15][16][17] method is applicable to message passing and shared memory programs. The method records the order in which messages are delivered in a preliminary execution. These traces are then used during re-execution to force each message to be delivered to the same operation as during traced execution. The method allows on-the-fly detection of racing messages.

The *Parasight* debugger[1] allows dynamic creation and linking of "parasite" programs into a parallel application. It uses a built-in dynamic loader to load and link the user's target program, and it builds a memory-resident symbol table which is used to link in the parasites. These are then able to interact with program memory and to create and control program instrumentation.

The *Paradyn* system [12] provides methods for dynamic insertion and removal of instrumentation in a running program. It can detect performance problems at run time. The inserted instrumentation can evaluate assertions on the fly.

As presented in this paper, *p2d2* is the first debugger to fully integrate trace visualization and a replay mechanism into a state-based debugger in order to provide consistent breakpoints and an undo mechanism for debugging message passing programs written in Fortran or C.

6 Conclusions

The display and use of a program's execution history can have a significant beneficial effect on the debugging process for a parallel message passing program. It can provide the user with an ability to see the big picture of execution and control the execution of a target program using it. We implemented history-based features in the portable distributed debugger *p2d2*. In particular, we added trace visualization, the ability to set consistent break points, as well as *replay* and *undo* operations.

In order to generate the program traces used by the history features, we used three instrumentation tools: the *AIMS* toolkit, a new tool (`uinst`), and the profiling interface to the MPI message passing library. These tools give the user a wide spectrum of possibilities for collecting execution history. They vary in the effort required, the performance overhead, the history event resolution, and portability. One conclusion we draw from this work is that a compiler-debugger interface could reduce the instrumentation effort and the performance overhead, while increasing event resolution, portability, and transparency to the user. Ideally, a presence of a command line option such as "`-i`" or even "`-g`" should cause the compiler to insert instrumentation calls to a debugger-provided library.

There is an additional instrumentation strategy which remains to be explored. The Paradyn system [12], and in particular its *Dyninst* API [`http://www.cs.umd.edu/~hollings/dyninstAPI/`] would permit debug-time instrumentation of the source code. If traced runs are always initiated by the debugger, this would free the user from any instrumentation concerns whatsoever. We need to experiment with the system to determine how conveniently it can be used for runs initiated outside of a debugger.

In our implementation, after trace information is communicated to *p2d2*, it can be displayed as a time-space diagram using either *NTV* or *VK*. The user can explore the trace, set a stopline and request *replay* or *undo* operations, in addition to the standard set of debugging operations provided by *p2d2* [5]. Efficient navigation of execution history is based on a graph abstraction of execution history. It allows fast location of events of interest in the program traces. It also provides a good basis for execution analysis for locating circular dependencies of messages and locating the missed messages and irregularities in message traffic.

Our current implementation of *replay* and *undo* is done in straightforward manner by re-executing until an execution marker threshold is encountered. We could improve on this by periodically checkpointing program states and keeping a logarithmic backlog of process states.

Our trace-driven debugging techniques are portable to a number of parallel platforms and are applicable to deter-

ministic message passing Fortran and C programs. In particular, they are applicable to single-threaded programs running without interrupts or message passing errors, which do not use volatile variables or the `MPI_ANY_SOURCE` wildcard, and which do not call `MPI_CANCEL` and `MPI_WAITANY`. An adaptation of instant replay techniques [7] would facilitate the removal of some of these restrictions.

The trace-driven debugging gives a user a clear, consistent way for exploring the execution of a message passing program and for controlling its execution during a debugging session. It reduces the effort which usually required for locating bugs in parallel message passing programs.

7 References

[1] Z. Aral, I. Gertner. High-Level Debugging in Parasight. *Proc. SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Jan. 1989, pp. 151-162.

[2] J. Cuny, G. 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.

[3] S. L. Graham, P.B. Kessler, M.K. McKusick. An Execution Profiler for Modular Programs. *Software-Practice and Experience 13*, pp. 671-685 (1983).

[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] E. Kraemer, J. T. Stasko. The Visualization of Parallel Systems: An Overview. *J. of Parallel and Distributed Computing 18*, pp. 105-117 (1993).

[7] 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.

[8] 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).

[9] Y. Manabe, M. Imase. Global Conditions in Debugging Distributed Programs. *J. of Parallel and Distributed Computing 15*, pp. 62-69 (1992).

[10] J. May, F. Berman. Panorama: A Portable, Extensible Parallel Debugger. *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993, San Diego, CA, pp. 96-106.

[11] 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.

[12] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth R. B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer 28* (11), November 1995, pp. 37-46.

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

[14] MPI-2: Extensions to the Message-Passing Interface. July 18, 1997.

[15] 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.

[16] 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.

[17] R.H.B. Netzer, B.P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. *Supercomputing 92*, Minneapolis, MN, pp. 502-511.

[18] S. Shende, J. Cuny, L. Hansen, J. Kundu, S. McLaughry, O. Wolf. Event and State-Based Debugging in TAU: A Prototype. *Proceedings of SPDT'96 SIGMETRICS. Symposium on Parallel and Distributed Tools.* May 1996, Philadelphia, PA, pp. 21-30.

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

[20] 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).