TITLE: TOWARDS HETEROGENEOUS DISTRIBUTED DEBUGGING

AUTHOR(S): Suresh K. Damodaran-Kamal, Jeffrey S. Brown, CIC-8, Los Alamos National Laboratory

SUBMITTED TO: Supercomputer '95, San Diego, CA, December 3-8, 1995

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes.

The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Towards Heterogeneous Distributed Debugging

Suresh K. Damodaran-Kamal       Jeffrey S. Brown
Distributed Computing Group
M/S B272, CIC-8,
Los Alamos National Laboratory,
Los Alamos, NM-87545
Email: {suresh,jxyb}@lanl.gov

Abstract

Several years of research and development in parallel debugger design have given us several techniques, though implemented in a wide range of tools for an equally wide range of systems. This paper is an evaluation of these myriad techniques as applied to the design of a heterogeneous distributed debugger. The evaluation is based on what features users perceive as useful, as well as the ease of implementation of the features using the available technology. A preliminary architecture for such a heterogeneous tool is proposed. Our effort in this paper is significantly different from the other efforts at creating portable and heterogeneous distributed debuggers in that we concentrate on support for all the important issues in parallel debugging, instead of simply concentrating on portability and heterogeneity.

Keywords: Debugging, Message Passing, Distributed Computing, Heterogeneous Computing, Portable Tools.

1 Introduction

Many factors have contributed to the current interest in portable and heterogeneous distributed tools. Availability of comparatively more affordable yet powerful workstations has resulted in the transfer of more and more program development work to the workstations from more powerful computers. Traditionally, these workstations used to be the front ends for supercomputers (currently Massively Parallel Processors (MPP)) running at the back end. This situation lead to the development of debugging tools on the workstations that have the
same interface as the supermachines. For example, the debugger called TotalView[9] on Cray T3D is also implemented on SunSparc workstations. Thus, the user can develop the code on the workstations, avoiding the more costly alternative of developing on the supermachine. However, the front end debugger is not a distributed debugger. Therefore, a user does not have the option to develop, debug, and run distributed programs on the work stations, and later port the code to the massively parallel processors. If the distributed debugger were portable and could work on a cluster of workstations, then the user would have had this option.

Another factor has also contributed to the need to have portable tools. Today’s most powerful machine may not be the most powerful machine of tomorrow at a particular establishment. Much effort is invested in creating working programs on a particular machine. Porting these codes to another machine with dissimilar architecture can be a nightmare. In the process, the user is forced to learn a new set of tools for the new machine. If the tools are portable, and designed in such a way that the user interface component can be kept the same, then the transition can be made easier. For example, CM5 has a debugger called Prism[49]. However much a user may like this tool, we do not have a port of Prism on, say, Cray T3D. Thus, lots of user effort can be saved if there are portable tools.

There is yet another factor that contributes to the need for heterogeneous tools. Some of the users have come to use more than one type of machine to get their programs rolling. There are economic reasons (some machines are more costly), and performance reasons (some machines do certain jobs better) for this user behavior. The availability of message passing libraries such as PVM[48] that will work on heterogeneous platforms has made this user behavior more prevalent. These users are not satisfied with portable tools alone. They require tools that will work on the different processes of the same program when these processes are running on heterogeneous platforms.

The development tools on distributed message passing environments fall basically into two categories: correctness debugging tools that are used for eliminating functional errors, and performance debugging tools that are used for tracking down performance bottlenecks. With respect to implementation, there are two broad categories of parallel tools: trace based tools, and run-time (or dynamic) tools. Trace based tools such as ParaGraph[23], Xpvm[20], AIMS[51] can be used after the completion of execution (postmortem analysis). Some of the trace based tools such as Xpvm[20] can also be used as an execution monitoring tool. Trace based tools are more useful as performance debugging tools, rather than correctness debugging tools. We concentrate in this paper on run-time correctness debugging tools.

The currently available correctness debugging tools for distributed message passing machines are not satisfactory enough, though users may be interested in some of the features each tool has to offer. As a result, these tools are underutilized. Therefore, programmers
continue to use print statements instead of debuggers. Even those who use tools tend to use homegrown tools[44]. There is a prevalent notion that tool developers do not understand the requirements of the users. The Parallel Tools consortium is an attempt to fill in the communication gap among the tool developers, researchers, and users[43].

Our interest in the development of distributed debuggers that are portable and will work on heterogeneous platforms stems from interactions with our user community at Los Alamos National Laboratory. The programs that our user community develops are mostly scientific programs related to computational fluid dynamics, numerical particle simulation, Monte Carlo simulation, and lattice quantum chromodynamics. In our attempt to build a portable and heterogeneous tool we carefully consider the user demands, as well as the current technology.

Though there is a demand for heterogeneous distributed tools, the industry has not yet come out with any production quality tool that can claim to be heterogeneous. This may be because vendors are not loosing sales of new systems due to inadequacies of their debuggers[7]. Consequently, less resources are allocated to debugger development. Vendors of specific machines are not much interested in making their tools heterogeneous since traditionally a specific vendor did not see much profit from making the tool heterogeneous. In addition, there is a wide variation in the parallel programming environments that makes the prospect of making a tool work with different programming environments a nightmare. The recent efforts at standardizing the parallel message passing environment by defining MPI[17] may help reduce this problem.

The technologies to enable the development of a really useful heterogeneous distributed debugger already exists. However, these technologies are spread out into different tools developed by different vendors, researchers in laboratories, and academia. Tool development will be easier if there are well defined protocols for interaction among the components of a tool, the compiler, and run-time systems. An understanding of user requirements, the technical issues, and technologies to resolve these issues will lead us to the definition of what should constitute a portable and heterogeneous distributed debugger. Further, the identification and definition of parts of such a debugger will encourage third party tool developers in industry and research to develop the components of the debugger in collaboration. Also, identification of the hooks required in the message passing environments to support such a debugger will lead to better integration of the debugger with the environments.

There are several techniques to deal with the issues of process control, nondeterminism, scalability, data visualization, etc. both from industry as well as academia. Our attempt in this paper is to analyze which of these techniques can be useful from the user’s point of view, and to propose an integrated approach to the problem of designing a heterogeneous, portable distributed debugger. In particular, our goals in this paper are the following:
• investigate the issues in heterogeneous distributed debugging,
• identify the features of a useful heterogeneous distributed debugger,
• investigate how the existing technology can support these features,
• propose an architecture for a heterogeneous distributed debugger incorporating these features, and
• identify the support required by the compilers and run-time system for such an architecture.

In Section 2, we describe the related efforts at creating heterogeneous or portable distributed debuggers. The issues in distributed debugging are discussed in Section 3, along with the issues in making a debugger portable and heterogeneous. Our proposal for a heterogeneous distributed debugger is in Section 4. Section 5 concludes the paper.

2 Related Works

One of the early works on heterogeneous distributed debugging for event detection through Event Based Behavioral Abstraction (EBBA) is by Bates[1, 2]. The abstracted view of events occurring in heterogeneous processors is supported by a EBBA based toolset. This technique has been applied for run-time detection of event patterns specified at a higher level of abstraction. BEE system[6] is a portable platform for building distributed event environments to monitor and debug the performance of heterogeneous applications in a network computing environment. Event detection is not the primary focus of this paper, though we look into the related issue of watch point implementation is Section 3.7 as part of our general goal of debugging concurrent scientific programs.

A client-server based organization for achieving portable and heterogeneous parallel tools is discussed by Cheng and Hood[7, 8]. Their tool, p2d2, deploys debugger servers at the machines where processes are run. The user interface to p2d2 is configured as a client to these debugger servers. A principal theme in the design of p2d2 is that by defining a standard function call interface that each vendor must support, the responsibility of developing the debugger servers can be moved to the vendors. However, this project concentrates mainly on the issue of portability and heterogeneity, leaving out the other issues of parallel debugging such as process control, nondeterminism, etc. This paper proposes an integrated approach towards parallel debugging that addresses all these issues. The underlying mechanism used by p2d2 for communication between the client and a server is remote procedure call(RPC).

Panorama[37] presents another interesting approach to the design of a portable tool. Panorama supports visualization of a program execution occurring in remote machines.
Panorama interfaces with the native debugger, the debugger provided by the vendor on the remote machine, instead of deploying debugger servers like p2d2. Mapping of user commands given to Panorama into native debugger commands is supported using regular expressions[38]. This tool uses remote login as a way to connect to the native debugger.

IVD[22] is another tool aimed at providing data visualization and run-time debugging across heterogeneous machines. IVD maps the native debugger window interface buttons to IVD buttons, thus enabling a user to debug a program remotely using the native debugger.

Tools such as p2d2, Panorama, and IVD contain useful techniques to support portability and heterogeneity. However, there are other issues in parallel debugging such as program control, nondeterminism, user interface, and data visualization that also need to be addressed along with portability and heterogeneity. In this paper, our goal is to analyze the various techniques that are available to address these issues so that a portable and heterogeneous tool can be built to address these issues effectively. During our analysis of these techniques, we will discuss in detail the above-mentioned tools as well as other tools.

3 Issues in Heterogeneous Distributed Debugging

There are several issues involved in the design of a heterogeneous debugging tool. Some of these technologies are embodied in debugging tools. In this section we examine those techniques, and see how useful they are for heterogeneous distributed debugging.

3.1 Portability and Heterogeneity

Portability and heterogeneity are related issues in parallel tools development. A portable tool need not work in a heterogeneous environment, even though it is quite possible that the tool can be ported to each environment (machine) in the heterogeneous environment. A heterogeneous distributed tool, on the other hand, can be invoked on all platforms in a heterogeneous environment, and can have its components executing on one more of the platforms in the heterogeneous environment. It follows that portability of a distributed tool is a prerequisite to heterogeneity. A more limited form of heterogeneity can have the restriction that the distributed tool can be invoked from only a subset of all the platforms in a heterogeneous environment. Therefore, the issues of portability and heterogeneity are related, though distinct.

A native debugger is a debugger that runs in a machine, and is capable of manipulating the symbol table and run-time properties of the process(es) being debugged in that machine. The native debugger may be either sequential or parallel. A major issue in incorporating heterogeneity in a distributed debugging tool is how much of the native debugger that a user interacts with explicitly. The aspects of a native debugger that the user has to be concerned
with are its user interface, command syntax, way of accessing process information, and techniques to control process execution. Currently, there are broadly three main approaches to deal with these issues.

One approach is to provide to the user the same user interface as the native debugger. Using X window system, this is accomplished by setting the display at the user’s terminal. The user is not allowed to communicate to the native debugger, except through the user interface of the native debugger. A main difficulty with this approach is that the user is confronted with differing user interfaces of the native debuggers of the different machines. Some of these difficulties in differing command interfaces can be resolved by providing a uniform command line interface and mapping the commands to the native debugger automatically[38]. Another difficulty with providing the same interface as the native debugger is that co-ordinating the debugging of multiple processes has to be done manually, unless causal break pointing is used as in Xmdb[11, 13] (also see Section 3.4).

Another approach is to map the functions of the native debugger to the buttons in a standard user interface provided by the distributed debugger. Again, the user can communicate to the native debugger only through the buttons mapped to the distributed debugger buttons. This is the approach used in the IVD project[22].

Yet another approach is based on a client-server architecture for the distributed debugger. In this case, a native debugger runs as a server for the front end of the distributed debugger that acts as a client. The user can communicate to the native debugger server through the user interface provided by the client. An important distinction from the previous approaches is that the access of the native debugger is transparent to the user, as communication to a native debugger is not done through the visual interface provided by the native debugger. An example of this approach is p2d2 project at NASA, Ames[7]. A similar approach, though for homogeneous work stations, is used in the TotalView debugger from BBN Inc.[3, 4]. A disadvantage of this approach is that a debugger server needs to be implemented for every machine. This server may interface to an existing debugger, or may implement own debugging capabilities on processes. Either case requires substantial programming effort from the implementor. It is possible to reduce this programming effort if the vendors provide the debugger servers.

The interface to the debugger may be command line, window based, or both. A window based interface is definitely easier to use than a command line interface. However, there are users that use the computing resources remotely, but cannot display an X window system based user interface. In such cases, a command line interface is necessary. There is currently no agreement as to what constitutes a standard user interface. One approach could be to use an interface that users are familiar with, such as dbx[33] or gdb[47]. This approach may make it easier for sequential programmers to move into a parallel environment. Parallel
debuggers such as Prism[49], p2d2[7], and TotalView[3] have, however, their own unique user interfaces.

3.2 Programming Environment Support

Parallel programs are written in different languages. Some of these languages hide the message passing that goes on to make the programs work. An example is High Performance Fortran (HPF)[24]. Yet some other programs are written in conventional languages such as Fortran 77, but use parallel libraries such as ScaLAPACK[16] to hide the message passing complexity. Similar approaches based on object oriented programming languages such as C++ is becoming popular too[35, 46]. At the other extreme are programs that explicitly use message passing libraries such as PVM[20] or MPI[17]. Though it may be that a single program may not use all of these programming approaches, a heterogeneous debugging tool has to support debugging programs in all these categories. The programming environment needs to support some mechanisms so that a distributed heterogeneous debugger may work in the programming environment. The issues involved in supporting debugging in a programming environment are the following:

- how can the debugger provide information at the level of abstraction of the program,
- how to inform the debugger the distribution of program data,
- how should the debugger gain access to the symbol table and other process information, and
- the start-up problem.

We shall illustrate these issues based on the HPF and PVM programming environments.

In some implementations, HPF programs are first translated to Fortran 77, and then compiled by a Fortran compiler[45]. Currently, a clean mapping of the translated Fortran source to HPF source does not exist. This situation forces an HPF programmer debug in terms of Fortran 77 code. Moreover, the HPF compiler decides where to place the user data in the distributed machine configuration. The debugger needs to know this initial distribution of user data. Another problem is the provision in HPF for run-time redistribution and realignment of data. There must be a mechanism to let the distributed debugger know the run-time movement of data. The problem of attaching a user process to the debugger before the user process executes any user code is referred to as “start-up problem” in [?]. PVM 3.3[20] provides a tasking mechanism so that a user process can be spawned under a debugger. Such support by a message passing environment is useful for integrating a debugger effortlessly with the environment.
3.3 Nondeterminism

Many parallel programs, especially those that are not yet debugged, display nondeterminism. Usually nondeterministic behavior occurs in programs using explicit message passing mechanisms. However, even when the message passing is implicit, possibility of race conditions occur when the programmer chooses to override the safety features provided by the programming language for better performance. Typically, attempts to improve performance by hiding communication delays leads to nondeterministic behavior.

When a program is nondeterministic, it is possible that a program will behave differently in different executions, given the same input. Nondeterminism makes debugging parallel programs much more difficult than sequential programs. Probe effect, unrepeatability, and race conditions result from the nondeterministic behavior of programs. We shall discuss each of these problems, and the current techniques to combat these.

3.3.1 Probe Effect

Probe effect[19] refers to the fact that observing the behavior of a nondeterministic program may alter its behavior. This effect is observable in trace based tools as well as in dynamic tools. For example, insertion of a print statement in a parallel program can potentially change the behavior of a parallel program, if the program is nondeterministic. When you insert a print statement in a program to observe the value of a variable, you may be observing the behavior of the program with the print statement.

Since we are primarily interested in dynamic tools, let us explore the techniques to remove the probe effect. The impact of the probe effect is that even a break point inserted in the source code can alter the behavior of the program. It has been shown that an OtOt (One-Thread-at-One-Time) strategy of debugging helps to avoid the probe effect[10, 14]. The OtOt strategy consists of debugging with respect to one process in a program such that the execution of only that process is controlled. The debugger for PVM, called Xmdb[11, 13], implements this strategy of debugging.

3.3.2 Execution Replay

Nondeterminism also causes unrepeatability, i.e., difficulty in reproducing the behavior of a parallel program. Sequential programs may be debugged by cyclical debugging. However, it is difficult to debug parallel programs using cyclical debugging due to their unrepeatability. Execution replay, originally proposed by Leblanc and Mellor-Crummey[31] is a technique used to reproduce the execution of a parallel program. In execution replay, a program is re-executed using an execution trace. The execution trace is used to trigger and veer the re-execution of the program to match with the original execution. A technique for execution
replay for message passing parallel programs is described in [32]. In [32], integration of program replay and data visualization is discussed, so that the probe effect of tracing and visualizing large data can be minimized.

An implementation of program replay for PVM programs done at Hewlett-Packard Laboratory is reported by Mackey [34]. This implementation requires a change of the message passing library, as well as the PVM daemons. While replaying, it is useful to allocate the same process identifiers to the processes as in the original execution, if messages use process identifiers for communication. A technique that does not require a change in the message passing layer can be very attractive.

An interesting question regarding replay is whether the re-execution must match the original execution of all processes, or a subset of the processes. The implementation by Mackey chooses to replay the original execution in all processes. On the other hand, the replay mechanism in Xmdb [11] forces the replay to match the original execution with respect to a single process only. Though simple to implement, this approach to replay may not work in some cases, especially when the other processes do not succeed in generating the message that is consumed in the correctly replayed process. However, in Xmdb, the replay mechanism is combined with techniques to detect nondeterminism so that one gets forewarned about such situations. One the other hand, to do deterministic replay of all processes involved in a computation, it is necessary to trace time-stamped messages in all processes.

The size of the trace data generated can be large for large programs. For message passing parallel programs, a technique to reduce the trace data is discussed in [41]. Since the process of taking a trace can induce the probe effect, the size of trace data is an important issue. A key requirement for reducing the trace data in [41] is the detection of race conditions. Thus, combining race detection and tracing (for replay), is advantageous.

3.3.3 Race Conditions

Race conditions occur when the behavior of a program depends on the order in which messages are received in a process. In this section, we discuss the techniques for specifying and detecting undesirable race conditions.

Most scientific programs are required to give the same result every time the program is executed, given the same input. When a program is also deterministic internally, detection of a race condition indicates an error. However, internally the behavior of many scientific programs is nondeterministic. Many times, it is convenient to program if nondeterministic behavior is allowed. For example, if the algorithm requires summation of a set of partial results produced by some processes, then the order in which the messages containing these partial results are processed at the summing process is immaterial. Therefore, a programmer need not impose an artificial restriction that the partial results can be added only in a certain
order. Yet, in some other situations, nondeterministic behavior is used to improve performance. In the previously mentioned partial results summation example, a restriction on the order of summation may result in performance degradation. Assume that the summation is part of an iterative loop: after sending the partial result, the process that sent the partial result will receive the next quantum of work. Now, it may happen that a process generating the partial result sends the message containing the partial result fairly early, but has to wait till its turn comes in the summation order. This waiting is an avoidable performance degradation. Thus, race conditions are not always erroneous, and sometimes are even introduced deliberately.

A technique for detecting race conditions in message passing programs is described by Netzer and Miller[41]. Their technique requires vector time stamps that are embedded in messages. Another technique for race detection that does not require time stamps is given by Damodaran-Kamal and Francioni[12]. The technique by Damodaran-Kamal and Francioni can also be used for detecting only undesirable race conditions[14] using a specification language[10, 15]. Xmdb[11] implements these race detection techniques for PVM programs.

3.4 Program Control

A parallel program running on a distributed memory environment can have many processes. Some of these processes may share the same source code. In order to debug a program, a user need to control the execution of a program. In a sequential program, one may put a break point at one or more places in the source code, however, a program will stop execution at only one break point. In parallel programs, however, there are more than one process, and consequently, more than one break point. Chasing the execution of a parallel program from one break point to another can be very taxing. The increased difficulty in debugging a parallel program in the presence of multiple break points is called the maze effect[?]. Two techniques are used to counter the maze effect. First is to group processes in such a way that the process group makes sense with respect to the run-time environment. The second technique is to use causal break points. We discuss both of these techniques below.

3.4.1 Process Structures

In a parallel program, when there can be several processes, it is convenient to group these processes into groups so that debugger commands can be given to all the processes in a group. If the debugger does not recognize these structures, then the user is relegated to the one-process-in-a-window approach to debugging. The debugging commands to each of these processes have to be manually delivered by the user, a serious problem to scalability. The relationships among processes, and the structures that evolve from these relationships are
Three types of structures may exist among the processes of a program: an invocation tree, communication groups, or code sharing groups. The processes of a program that are running under a message passing environment are related to each other through an invocation tree. In an invocation tree, the nodes are processes, and the children of a node are processes spawned by the node. Note that an invocation tree may not be the way a programmer would like to reason about the process structure.

Processes may join groups to support collective communication[20, 17] by an explicit call embedded in the program. It is possible to send broadcast messages to all the members of a group, as well as require all members of a group to wait at a barrier to synchronize. Intel introduced program contexts[27] to provide program grouping. Several parallel debuggers support similar grouping of processes[40, 28, 3, 49]. PVM 3.3 and MPI support the creation of dynamic message groups of processes[20, 17].

However, the members in a group do not necessarily have to share source code, though they may do so in SPMD (Single Program Multiple Data) model computations. Therefore, it is more appropriate to call the group of processes that share code as code share groups[3]. The pros and cons of different techniques to break the execution of a parallel program, and the role of the process structures in executing break points are discussed next.

3.4.2 Break Points: Where, How

A break point in a sequential program has a clear definition: a point at which the execution of a program will be stopped, and control of the program passed over to the user. However, a parallel program has multiple threads of control. In a message passing program, there are several processes, some of which may not even share the same source code. This scenario leads to several interpretation of a parallel break: point. In this section we will describe and analyze the different notions of break points.

Barrier Break Points: The first kind of parallel break point is what we call a barrier break point. A barrier break point occurs only in a set of processes that share the same source code. Another requirement for a barrier break point is that all such processes will reach this break point at some point in their execution. An SPMD (Single Program Multiple Data) program is likely to have one or more barrier break points.

For example, in a program written in a data parallel programming style\(^1\), it is possible that all processes will execute the statements in a for all loop concurrently. If there are no conditional statements inside the for all loop, then all the processes will have to execute all the statements inside the loop. A barrier break point at any of those statements will force

\(^1\)Though it is more natural to think programming in languages like HPF as data parallel, it is also possible to write such programs using message passing libraries such as PVM.
all processes to stop execution at the same point.

Barrier break points are useful not only in programs written in data parallel style, but also in programs that have processes that synchronize frequently. For example, it is possible that all processes will hit a barrier break point after executing some segment of code that involves communication among the processes. Normally, a consistent global state (a combination of the states of the individual processes) is conceivable, and may be verified at a barrier break point. Existing parallel debuggers support parallel break points in some form or other[49, 3]. Visualizing program data may be appropriate at a barrier break point, as it is possible for the user to reconstruct the state of a computation at a barrier break point.

Barrier break points are not sufficient to debug all the problems even in programs written in data parallel style, if there are conditional statements inside the scope of a forall statement, and the processes execute different branches of the conditional statements. The problem is that a break point on one branch of the conditional statement (inserted in all processes) is not enough to stop all processes at the barrier break point. In this case, what is the best way to stop the processes that do not execute the branch where a break point is inserted?

One solution is to send a signal to the processes that did not reach the inserted break point such that the processes will be stopped wherever they are in computation. However, these signals should be sent to all the other processes as soon as one process hits the break point. This is necessary, since otherwise it is not easy for the debugger to know how many processes will actually reach the break point. This is the default solution adapted by TotalView[3]. This solution is not adequate, since the user will face difficulty in reconstructing a global state with processes stopped at arbitrary points in computation.

Another solution is to let the processes that did not reach the inserted break point to continue execution until they cannot continue execution due to dependency on messages from the processes that are already stopped. This solution is more helpful to the user in reconstructing the state of the computation, since the user has at least the knowledge that either a process is at the inserted break point, or is waiting for an input from one of the stopped processes.

Another problem with barrier break points is that if the processes stopping at a barrier break point have messages sent to them that are not yet received, then reconstruction of the computational state also needs to take into consideration the messages in transit. In many cases, such reconstruction is quite difficult.

Thus, we conclude that barrier break points are useful in some situations, but not always. Causal break points, to be discussed next, can be helpful where barrier break points are not.

Causal Break Points: Another kind of parallel break point is based on the causal dependencies among the processes. In message passing programs these dependencies take the form of messages. If we break the execution of a program at a break point, then all the processes
that causally depend on the stopped process will stop at the receive statements for these messages. Such break points are called causal break points [18]. Run-time debugging based on causal break points is implemented for PVM programs in Xmdb [11, 13].

When stopped at a causal break point, calculation of computational state is relatively easier than barrier break points, as we are sure that the process is stopped because it can’t proceed without receiving further messages. In Figure 2, the processes are stopped as indicated because $P_2$ is waiting for a message from $P_1$, $P_3$ is waiting a message from $P_2$. The message from $P_3$ to $P_2$ is not received in $P_2$, and is a message in transit. However, calculation of the process state of $P_2$ does not require knowledge of the content of that message since $P_2$ is not scheduled to receive that message until later.

In cases where different statements inside a forall loop are executed by different processes, causal break points are useful. However, inducing causal break points with more than one manually inserted break point can lead to confusion as to which of the manually inserted break points caused the causal break point. A technique to test a parallel program for undesirable race conditions using causal break points is given in [14].

### 3.5 Data Visualization

Scientific parallel programs typically manipulate large amounts of data. The data may be available in single or multidimensional arrays. The ability to view these data, either numerically, or transformed into different types of views is an important requirement for debugging scientific programs.

In distributed programs, the data to be viewed may be distributed across many machines. The main issues in data visualization are the following: obtaining array structure and decom-
position information, how to specify the data to the visualizer, what views are most useful, and how to provide easy navigation across multidimensional data. An effort at designing a distributed array query and visualization tool is initiated by Malony et al.[36]. Prism[49] is a debugger that provides good support for array visualization. Prism supports the display of array data in color maps, surface contours, and also in numerical form. The TotalView version for PVM support[4] provides numerical array display. It is worth mentioning that the commonly available dbx[33] debugger, and the window interfaces to dbx such as dbxtool, debugger, etc. do not provide good support for displaying Fortran arrays.

3.6 Scalability

Scalability of the debugger is an important issue. There are two unconnected issues in scalability. The first is how effective will the debugger be with large programs. The second issue is how the debugger will deal with large number of processes. We discuss these issues below.

Program Size: When large programs are being debugged, then it is possible that a program crash occurs after several days. It is not practical to start a program from the beginning so that the problem can be diagnosed with a debugger. Therefore, it is necessary to have check pointing facility integrated with the debugger, or the programming environment. Such check pointing will enable the restarting and debugging of the program from the last place where the check pointing is done.

Number of Processes: When the number of processes becomes large, then one-process-in-a-window approach will not work well, and it is worth looking at alternative approaches. One solution is to allow more than one process to share a window, as discussed in Section 4.3. Such a solution may also be combined with process grouping. Another solution is to debug processes based on the message flow[10]. In this case, a program is debugged with respect to one process, and the other processes are debugged based on the message dependencies on this process. A combination of these differing approaches may prove to be even better.

When we are on the topic of scalability of debugging, it is relevant to comment on single processor debugging of parallel programs. Though this approach may be the only choice when one does not have a parallel debugger, it is worth noting that there are many instances of error that may appear only when running on a parallel environment, sometimes only in a heterogeneous environment. A typical example is timing error. Sometimes, it is recommended that it is good to debug a program for “serial” errors as a prelude to debugging the program for “parallel” errors[20]. However, often the “serial” errors may induce “parallel errors” or vice versa.
### Table 1: Capabilities

<table>
<thead>
<tr>
<th>Features</th>
<th>Prism</th>
<th>BBN TotalView</th>
<th>CRI TotalView</th>
<th>p2d2</th>
<th>IVD</th>
<th>Xmdb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Control</td>
<td>barrier</td>
<td>barrier</td>
<td>barrier</td>
<td>manual</td>
<td>manual</td>
<td>causal</td>
</tr>
<tr>
<td>Data Visualization</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Race Detection</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Checkpointing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process Grouping(^a)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Watchpoints</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\)Different debuggers differ in their support.

### 3.7 Watch Points

In a sequential program, a watch point is used to “watch over” read/write access to memory locations by a program. Whenever a specified variable is accessed in a specified way (read/write), or when a predicate involving these variables evaluates to true, the execution of a program is stopped. An efficient implementation of watch points for Cray X-MP and Cray Y-MP machines in Los Alamos Debugger(1db) is described by Brown and Klamann\(^5\). In parallel programs, such predicates involve variables in more than one process. Whenever a predicate becomes true, the execution of the program is stopped.

In order to break the execution of a distributed program, first it is necessary to detect a global state (at which the processes are supposed to be stopped), and then stop the processes before their states change. For example, let us say that a program is required to stop in the following situation: \(A[*] < 0\), where \(A\) is a single dimensional array, and \(*\) indicates any element of the array. Let the array be distributed among 4 processes. To detect this condition, each process has to detect when the value of any element of \(A\) becomes negative in that process. Once this situation is detected, the processes have to be stopped. However, stopping the other processes is hard, since there is always an arbitrary time delay between the detection of a condition and the stopping of individual processes. One technique to overcome this problem is to stop each process at the same virtual time instant such that there is no \(happened-before\) relation \(^{30}\) relation among the states of the stopped processes. An algorithm to stop a distributed computation at a break point is given by Miller and Choi \(^{39}\). This algorithm is useful when a global predicate based on many process is defined, and the processes are stopped as defined by this global predicate. Flowback analysis\(^8\) may be used to reconstruct the events after the processes are stopped, so that the precise reason behind the negative value in array \(A\) can be determined.
Table 2: Platforms (current status)

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Prism</th>
<th>CRI TotalView(^a)</th>
<th>BBN TotalView(^b)</th>
<th>p2d2</th>
<th>IVD</th>
<th>Xmdb</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunOS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Solaris</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM AIX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>HPUX</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cray T3D</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM5</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\)CraySoft version.

\(^b\)PVM version.

In this section we have discussed the issues involved in the design of a heterogeneous distributed debugging, and the current technology to resolve these issues. A summary of some of the features of existing debuggers is given in Table 1 and Table 2. In the next section, we give our proposal for a heterogeneous distributed debugger.

4 The HDD Proposal

In this section we present our specifications for a heterogeneous distributed debugger (HDD). We present an architecture for HDD and discuss how this architecture can satisfy our requirements. The infrastructure support required from compilers and run-time systems are also discussed.

We assume an underlying message passing environment (MPE) such as PVM\(^{20}\), or MPI\(^{21}\) running over the heterogeneous machine. Note that when data parallel programming languages such as HPF are implemented over an MPE, then this assumption is true.

In the interest of increasing the portability of as much code as possible, we decided to adopt a client-server approach similar to that of p2d2\(^{7}\). Another decision in the design of HDD is to follow an object oriented approach. We call the process being debugged a debuggee process. A native debugger is a debugger that is working at a machine, and which controls the debuggee process. In a client-server approach, the native debugger is wrapped up inside a server to the HDD client(Figure 2). An HDD client can be connected to more than one debugger server, and each of the debugger servers can have more than one process running under it. If the machine is an MPP, then there can be many debuggee processes, each in a processor. The user interface to HDD need not necessarily reside at the machine where the
HDD client is running. Note that a native debugger may have user interface components that are useful in shedding light on the unique aspects of a specific machine. In such cases, debugging actions may need to be done through both the native debugger user interface, as well as the HDD client interface. The HDD interface to the user process allows only one process in a window, though support for commands to group of processes is also provided through the same interface (see Section 4.2). The various aspects of the HDD proposal such as the start up problem, process control, functionality of a debugger server, data visualization and querying, techniques to combat nondeterminism, watch points, scalability are discussed in this section.

4.1 Start Up

An important aspect of deploying the debugger servers is the way they are integrated to the underlying message passing environment. When a process is created under an MPE at a machine, then the debugger server at that machine needs to be aware of this process. This problem has been referred to as the start up problem (see Section 3.2).

When an MPE is started on a virtual machine, the machines over which the MPE will be running are either already specified, or machines will be added later on. When the machines over which the MPE will be running is already specified before initiating an MPE over a set of machines, we can assume that an MPE daemon is already running at that machine(Figure 3). In this case, the HDD client starts up a debugger server as a process of the MPE. Note that the HDD client is also a process of the MPE. The advantage of creating HDD under the MPE is that the debugger servers can now utilize all the services that are offered by the MPE. However, if the debugger servers need to communicate faster, they may
do so without using the communication facilities offered by an MPE. Thus, we combine the best of both worlds.

The debuggee processes are spawned by the debugger servers. Note that this does not happen by default. The MPE has to provide a function that will defer the spawning of tasks to the debugger server. PVM provides the `pvm_reg_tasked()` function[20], though currently MPI does not provide such a function\(^2\).

When machines are added to the MPE after it is already running on a set of machines, we require that the addition of newer machines is to be done through the HDD client so that the debugger server can be started up on the newly added machine. PVM conveniently provides `pvm_addhosts()` function to support this, though MPI does not provide an analogous function. The HDD client or the debugger servers can enquire the current configuration of the MPE (`pvm_config()` in PVM).

### 4.2 Process Control

The user processes running under the MPE form structures, and these structures are useful for better program control, in particular in setting break points. We discuss these process structures, the different ways of setting break points in HDD, and message based debugging in this section.

\(^2\)We are currently arguing for the inclusion of this and other functions useful for integrated tools in the MPI standard.
4.2.1 Process Structures

Three types of process structures, i.e., invocation tree, communication groups, and code sharing groups, are to be made available to the user to specify commands that pertain to these structures. We do not permit more than one process to share a window. However, commands to all members of a share group can be given from the window of a process (see discussion on Code Share Group below). We describe next the debugging commands to each of these specific structures required to be implemented in HDD.

Invocation Tree: The HDD commands that are directed at a node in an invocation tree are kill, show, unfold, fold, and add code share group. When the kill command is directed at a node, the process corresponding to a node, and all the children of the process are killed recursively. The show command will display where the process corresponding to that node, and all its children (recursively) are executing, and their status (running, blocked, suspended). This command will also display the communication group and the code share group each process is part of. The unfold will display all the immediate children of a node, and fold will stop displaying the children of the node. Add Share Group command will add the specified node to a specified share group.

Communication Group: A communication group is always formed with calls to the message passing environment by the processes. The following commands are used on a communication group: show (to show the members of the group, and the execution status of the members), kill (to kill the members of the group), create code share group (to create a code share group out of all the processes in the group), stop all (to break the execution of all processes in a group), and block (to block delivery of messages to the processes in the group). The create code share group command is useful when all the elements of a communication group share the same code. This command is particularly useful when debugging SPMD programs. The stop all command is useful if the processes do not reach a barrier synchronization point, and the user wants to find out where the processes are (possibly) hanging.

Code Share Group: A code share group contains processes that share the same code. To create a share group with one process, create code share group command can be used. Processes may be added or deleted from a share group by add share group, or delete share group commands. A code share group can be deleted using a delete code share group command. The following commands are used on a code share group: show (to show the members, and the execution status of members), kill (to kill members of the group), insert break point (to insert break points in all processes in the group at the same place in the source code), delete break point, step (to single step in all processes in a group when at a break point, next (to step over the same function in all processes, continue (to resume execution in all the processes of the share group), stop all (to stop all processes in the share
group), and block (to block delivery of messages to processes in a share group). Note that to insert and delete break points, the user has to look at the source code. When the user is setting a break point in the source code of a process, he/she can specify that a break point will be set at the same place in all the processes in the process's share group (if the process is a member of a share group). Similar specification is done for step, next, and continue.

4.2.2 Break Points

We permit the execution of processes to be controlled in the following types of break points: manual, collective barrier, single causal, and collective causal. To set break points manually, the user may open one or more processes manually, by bringing up a window corresponding to the process. This is not a very attractive way of setting break points, due to the following reason. When inserting break points this way, it is natural to keep multiple windows (one for each process) open, leading to confusion on what is going on in each of the windows.

To set a collective barrier break point, the processes must be in a code share group. Then a break point can be set in all the processes in a code share group. Since there is only one process corresponding to a window, the user gets back control to a member of the code share group when the process hits a break point, when the user executes a stop all, or when the user interrupts the process. The user may check for the status of the other members of the share group while waiting for a process to reach a collective barrier break point.

The other types of break points are causal break points. A single causal break point is induced by blocking the messages to a single process, and allowing other processes to block waiting for messages from the already blocked processes. A collective causal break point is induced by blocking the message delivery to more than one process. It is also possible to block delivery of messages to all the processes in a code share group or communication group.

The causal break points are effective when combined with message based debugging. We specify the commands for message debugging next.

4.2.3 Message Based Debugging

Message based debugging is done on processes when one wants to experiment with the delivery of different messages to the processes. Message based debugging is in particular useful in explicitly message passing environments. It is possible to combine automatic race detection with message debugging as in Xmdb[11]. Display of messages in queue at processes is supported in Xmdb and the Intel IPD debugger[26]. The messages that are delivered to a process are not delivered directly to the process while doing message debugging. Instead, these messages are delivered to a message queue (MQ). The messages in the MQ are delivered to the process under user control.
The commands that can be applied to an MQ are listed below.

**Deliver:** There are three versions of this function. By default, this function will deliver the message at the front of MQ. Invoked with a pointer to one of the messages, this function will deliver the message that is pointed to. This function can also be invoked to deliver all the messages currently in MQ.

**Show:** This function supports the display of messages in the MQ. While displaying the messages, it also displays message contents, if so requested.

**Stop If:** Stop when a message with a specified attribute arrives, or more generally, when a message expression[^15] evaluates to true.

To support message based debugging, a mechanism has to be provided that will block the messages from being delivered to the debuggee process. This could be supported by a change in the library calls made to the MPE for receiving messages, or by instructing the MPE to deliver messages intended for the debuggee processes to the debugger server. Let us consider both these approaches. The first approach requires changes to the source code of the MPE - not an attractive choice. However, this approach will reduce the additional complexity of involving the debugger server in the delivery of messages. The second approach requires support from the MPE to deliver messages to the debuggee process to the debugger server. The debugger server may send these messages to the correct debuggee processes under user control. This approach is conceptually simpler. We intend to pursue the first approach until support for the second approach is available with MPEs.

### 4.3 Debugger Server

The following functions need to be supported by a debugger server. We intend to pursue a layered approach to the design such a debugger server similar to [25]. When a debugger server is debugging on multiple processes in the same machine, many of the following commands will have to be qualified with a process identifier. In the absence of a qualifier, the commands may be applied to any of the processes known to the debugger server. We classify the debugger server commands into two broad categories: basic commands and message commands. Basic commands are the commands that one may give to a debugger when debugging a sequential program. The message commands are used to control the execution of a process by not explicitly inserting break points in the source code (see previous section). The basic commands are given below.

**Create Process:** This function creates a debuggee process. The HDD client is notified whenever a process is created under the debugger server. As the debugger server is itself spawned as a process within the underlying MPE, the information on the server is available with the MPE, and can be queried by the HDD client.

**Kill Process:** This function terminates the execution of the debuggee process. As in the
creation of a process, killing of a process also results in notification of the MPE of this event. **Attach Process:** This function is used when a process is already running, and needs to be attached to the debugger server.

**Detach Process:** To release control of a debuggee process from the debugger server, this function is used.

**Suspend:** This function interrupts the execution of the debuggee process and suspends its further execution until the user explicitly resumes its execution.

**Insert/Delete Breakpoint:** This function marks a specified location (statement) in the code so that when control is passed to this location, execution of the process is suspended prior to the execution of this statement.

**Insert/Delete Watchpoint:** This function specifies that the execution be suspended when specified memory locations are accessed in a specified way (read/write). It is also possible to define a function and specify that execution be suspended when that function evaluates to true.

**Catch Signal:** This function is used to suspend the execution of the debuggee process whenever it receives a signal. The debugger server gains control of the suspended debuggee process.

**Deliver Signal:** Sometimes the debuggee process requires signals delivered to them. Such delivery of signals is done by deliver signal function.

**Continue:** This function is used to resume the execution of a process from the suspended state.

**Single-Step:** The purpose of this function is to resume the execution of the debuggee process for one basic operation, and then suspend its execution. The basic operation is defined based on the semantics of the programming language. For example, a basic operation for a Fortran program is the execution of a statement (that is not a subroutine call).

**Read/Write Memory:** These functions allow reading memory contents, evaluating memory contents based on a given expression, as well as setting memory cells to some specific values. These functions also can be used to pipe the contents of the memory out of the process data space to an external tool, such as a data visualizer.

**Watch Memory:** This function will support detection of reading or writing (as specified in the command) of a variable, and alerting the user, optionally, based on the evaluation of an expression.

**Read/Write Execution Status:** The read function allows examination of the execution status of the debuggee process. The execution status contains the information on where the execution of the debuggee process will resume, and the resources used by the debuggee process. The write function can be used to alter the execution status.
4.4 Data Visualization and Querying

Data visualization as well as numerical display of multidimensional arrays need to be supported. However, at a time the user can look at only a 2D slice of the multidimensional array. The views supported are to be similar to that are available in the PRISM debugger\cite{49}, i.e., colormap, contour, threshold, dithered, and numerical. A 2D navigational palette to navigate in huge arrays will also be provided. The data visualization will be done only at a break point. We do not intend to support continuous display of data values.

It is possible to visualize data available in a single process, or distributed across multiple machines. When the arrays are distributed across multiple machines, the HDD client needs to be aware of the data distribution. It is possible that the user may wish to look at the data irrespective of where the data is situated. Another scenario is that the user may wish to observe the data located at specific machines. In explicitly parallel programs, the user is in control of data distribution. Therefore, the user has to notify where the data is located to the HDD client. However, for programs in HPF it is possible to get information on the distribution by using entry points in the HPF libraries. When run-time redistribution or realignment of data is done, the debugger server needs to be notified. This is done by stipulating that the the HPF routines that perform the redistribution and realignment call the communication library that will inform the corresponding debugger servers. Alternately, the user can manually inform the HDD client about the data distribution before requesting a data display.

The evaluation of expressions based on the contents of multidimensional arrays is also required. Such evaluation is also called array queries. Threshold visualization can help detect certain information on the contents of an array. For example, whether certain array elements are negative. However, if one wants to know the index of the array element that has the value 34, then array expressions are required. We have not finalized the design of the array expression as of now, but expect to present it in the final paper. We may also make use of the results of the distributed array query and visualization project\cite{36} initiated by Malony et al.

4.5 Nondeterminism

We can combat nondeterminism using two means: race detection and execution replay. A technique for automatic race detection using an OtOt strategy is given in \cite{14}. This technique allows specification of desirable races for explicit message passing programs, so that only undesirable races may be detected.

The algorithm given in \cite{14} can be adapted for detecting race conditions at the message receive calls in a debuggee process. Detection of race conditions is useful in that the user is
aware of the potential problems that can occur due to relative process execution delays or message delays. More details on race detection techniques to be used in HDD can be found in [10].

Execution replay will enable a user to repeat a given execution that is erroneous. A given execution does not repeat itself because of nondeterminism. When the nondeterminism is not erroneous, then it may be sufficient to replay for debugging purposes the program execution in any of the several (correct) ways, until the error is detected. Therefore, we first require the user to make sure that a given program does not have undesirable races using the race detection features. Subsequently, the user may replay a given execution with respect to a single process. Replaying with respect to a single process also reduces trace data significantly as the number of processes in a program goes up. These techniques are already implemented in Xmdb[11]. Fully deterministic replay integrated with checkpointing based on the work of Mackey[34] and Netzer and Xu[42, 50] is also to be incorporated in HDD.

Implementation of replay requires changes in the MPE. One change is to force the reuse of process identifiers of the original execution by the MPE during replay. This is necessary since the traced messages will contain the original process identifiers. For fully deterministic replay, it is also necessary to append time stamps with messages, requiring changes to the MPE library.

4.6 Watch Points

Traditionally, evaluation of watch points does not ensure quick response time, especially when the watch points are evaluated after the execution of every statement. In implementing distributed watch points, this remains an important issue. We propose evaluation of watch points at the entry and exit of functions only. This approach can work even in the case of optimized code[5]. In some cases, the only hope of debugging an optimized code (a normal requirement for large programs), is through watch points implemented through code patching.

A watch point needs to be specified before it is evaluated. The watch point specification will be based on the Los Alamos debugger[29]. There is an overlap of issues in implementing data visualization and watch points, hinged mainly on determining data distribution. The mechanisms in place for detecting data distribution for data visualization will work for distributed watch points too. However, for evaluating watch points correctly it is necessary to ensure that all data redistribution updates are available to the evaluator.

The watch points that we plan to implement must be discoverable locally at a machine. We do not support watch point evaluation that needs synchronization of multiple processes, as this process will lead to substantially slower execution, besides complicating the implementation. Whenever a watch point evaluates to true, the process where the watch point is
discovered is stopped, and the other processes in the program are sent signals to stop.

4.7 Attributes of HDD

We briefly discuss the attributes of scalability, portability, heterogeneity with respect to HDD. The infrastructure support for HDD required from the MPE is also discussed in this section.

HDD supports three types of process grouping. One can give debugging commands to all the members of its code share group from the window of a process. Therefore, one does not have the drawbacks that accrue from a process-in-a-window paradigm. Also, the ability to attach and detach a user process to the debugger server at will allows the user to concentrate on only the processes that the user is interested, instead of being forced to work with all the open windows. Support for checkpointing also makes HDD usable for large programs.

The portability of HDD depends on the portability of the HDD client. The HDD client does not use any machine specific code, rather relies on the debugger servers to deal with machine specifics. Thus, portability of HDD largely depends on the portability of the debugger server. In the design of the debugger server, a layered approach based on gdb[47] is expected to reduce system dependency. Heterogeneity of HDD also depends on the portability of the debugger server. Heterogeneity of HDD is limited by only the heterogeneity of the underlying MPE.

One may implement HDD by changing the underlying MPE. This approach is not attractive since changing the MPE will require a program to be run on the original MPE for normal execution and on the modified MPE for debugging purposes. We would like to debug a program that is running on a modified system. This goal requires that the MPE provide support for start up of the debugger servers, understanding the data distribution, for message delivery control, for replay, and for checkpointing. We have discussed the required calls required along with the related topic. Thus, it appears that integrated heterogeneous debugging require co-operation among the designers of the underlying message passing environment and debugger developers.

5 Conclusion and Future Work

Design of a heterogeneous distributed debugger is a complex task, even though we only consider message passing systems. We conclude that much of the various pieces of technology that go into the design of a heterogeneous distributed debugger that will work on real programs already exists. To prove the point, we have proposed the preliminary architecture of such a debugger. However for a such a debugger to exist the following prerequisites must satisfy:
• Co-operation and active dialog among the message passing environment designers and tool developers: in the absence of this co-operation the users will be forced to debug in an altered environment, instead of a production environment;

• Clear definition of the various pieces of the heterogeneous distributed debugger: such definition can encourage third party tool developers in the industrial and research communities which is necessary in the face of the resources required to implement such a debugger;

We are working towards the development of a heterogeneous distributed debugger based on this architecture at Los Alamos National Laboratory in collaboration with industry, academia, and other laboratories.

Acknowledgements

We wish to thank Robert Hood (NASA Ames), Doreen Cheng (Philips Research), John May (Lawrence Livermore), Al Malony (Univ. of Oregon), David Rich, John DelSignore (BBN Inc.), and Jim Kohl (Oak Ridge) for helpful discussions. The many users at Los Alamos who ventured forth with opinions are greatfully acknowledged.

References


[34] MACKEY, M. Program replay in PVM. In 1993 PVM User's Group Meeting (Knoxville, TN, May 1993).


