1 Introduction

IVD, Interactive V isualization Debugger , is intended to provide on-line and integrated mechanisms for debugging, performance analysis, and data visualization for message-passing parallel applications.

FIGURE 1. The overall environment

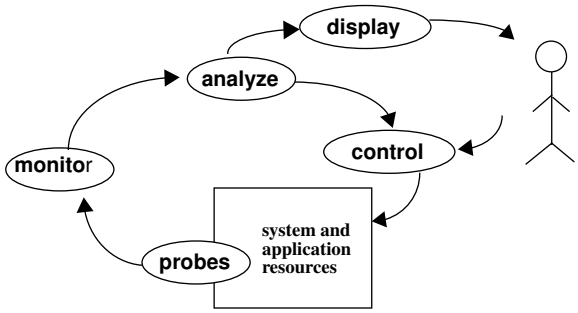


Figure 1 illustrates the overall environment in which the debugger will operate. The "probes" collect performance and computational data from system and application resources. The "monitors" aggregate and manipulate data from multiple probes before they are fed to the "analyzers". The "controllers" provide a feedback loop to adapt the application or system. The "display" modules provide data and performance visualization to the user . Our approach is to use existing sequential debuggers, performance analysis tools, and data visualization packages and to provide the essential glue to make them operate as described above. Where necessary , we will make minimal changes to the component tools and publish interface specifications that will allow other tools to be plugged in if they meet these specifications.

IVD is "on-line" in the sense that all debugging and visualization functions are available during program execution as opposed to afterwards. When this would introduce significant program perturbation, we use "program replay" in combination with the on-line tools. During an initial execution of a parallel program, the essential program behavior , consisting of message ordering and time-stamps of significant events, is traced with minimal overhead. During re-execution of the program, this trace is used to guide the program to the same behavior and to tag events with time-stamps from the initial execution. It is even possible during re-execution to collect more performance or application data and halt/restart various processes using the debugger without introducing any program perturbation.

Our tool will provide "integrated" debugging, performance analysis, and data visualization. Debugging, performance analysis, and data visualization can be synchronized. Processes in the parallel application can be halted by the debugger at the same point that performance and data visualization is being done. In addition, events from

some tools may trigger actions in other tools. For example, performance and data errors detected by performance and data analyzers may automatically cause the debugger to halt processes.

IVD will also provide parallel application checkpointing so that applications do not have to be re-executed from the beginning for cyclic debugging. They can be restarted from the last checkpoint before the point of interest. Without this feature, debugging long-running applications would be impractical.

Currently, IVD works with the PVM (Parallel Virtual Machine) message-passing library but it could be easily modified to support other message-passing systems. IVD does not support the full functionality described above now. The main accomplishments to date have been the following:

(1) IVD uses ESP, which is a novel mechanism to multicast window-based commands from a single control window to some subset of existing, unmodified debuggers and visualizers attached to various processes. Existing debuggers and visualizers do not have to be modified, recompiled, or relinked. ESP makes IVD extremely portable and allows heterogeneous tools to be used within a single debugging session.

(2) IVD provides program replay for PVM [2] programs by modifying the PVM library.

(3) IVD includes the capability to visualize application arrays distributed over the application processes. Arrays may be selected in an *ad hoc* manner at run-time and the design allows existing data visualization packages such as Gnuplot, Mathematica, or AVS to be used.

We describe each of these features in the following sections of the paper and provide some comparisons to existing parallel debuggers at the end.

2 Event Sense Protocols (ESP)

Current GUIs (graphical user interfaces) [1] are based on a single-threaded dialog, where the user operates on one single command button to invoke one application and to execute one single function at a time. There has been a need to have a mechanism to sense user commands and window events, and to control, manage, and multicast them to a set of selected multiple debuggers and visualizers for executing some functions simultaneously.

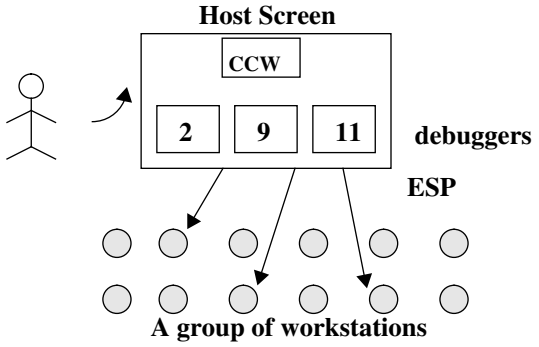
We want an interface that will support debugging functions for all selected processes using a single window. Such an interface will be very valuable for actions such as

simultaneous single-step execution in all selected instances. At present, PVM uses a separate window for each selected process. Users have to enter commands in each debugging window.

ESP solves the problem of how to sense user actions, to control, and to distribute input window events to each selected process' window for simultaneous execution in a distributed environment.

Figure 2 illustrates the host screen displaying a selected set of the debugging windows corresponding to a set of the running processes on different workstations. The host can control and manage the debugging steps for each running application through the IVD control window using buttons, text fields, *etc*.

FIGURE 2. A parallel visualization debugging session



2.1 Architecture Overview

ESP is built on a multiple client-single server model. There are two components we define in the IVD architecture: (1) A Concurrency Control Window (CCW); and (2) an Event Sense and Muticast Processing Manager.

FIGURE 3. An IVD architecture overview

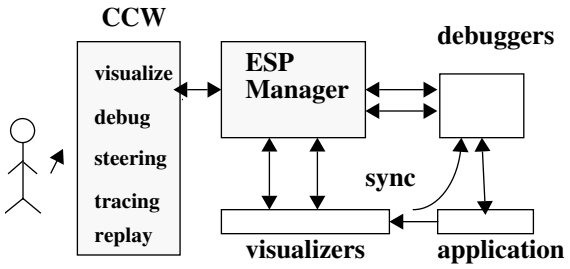


Figure 3 shows that IVD with ESP appears as a single visualization debugger to the user . ESP manipulates the user input events, such as keyboard and mouse, and then multicasts these events to each running visualizer and debugger. IVD automatically triggers the visualizers/debuggers to execute received events from the user. User

events are processed as if the window events had been directly entered into the visualizer/debugger windows.

2.2 ESP features

IVD uses ESP to integrate existing debuggers and visualizers to observe parallel application execution behavior in a distributed, shared-nothing multicomputer environment with the following features:

2.2.1 Heterogeneous processing

IVD is designed for heterogenous processing. Using ESP input event sense and multicasting capability enables IVD to access existing debuggers across various platforms to debug large, complex problems.

2.2.2 Scalability by grouping

IVD is scalable. IVD is built on a shared-nothing multicomputer environment. Each process has its own debug-ging and visualization facilities. ESP provides different "contexts" for the user to dynamically select the multicasting scope. By sending commands only to relevant processes within a context rather than all processes simultaneously , the user may also use contexts to limit the amount of overhead generated by running multiple processes.

With a context, the user can group certain debugger instances together . Each context then receives the same debugger commands entered on the IVD concurrency control window . By grouping the debugger instances, the user can indicate the execution order of groups of debugger instances. The user is not restricted to work only with individual debugger instances. This becomes useful when certain debugger instances are naturally associated with one another . For example, one context may contain all the slaves while another context contains only the master . Alternatively , there may be many contexts, each containing the debugger instances that are functionally related to one another . A given debugger may belong to more than one context or no context.

2.2.3 Portability

IVD uses standard X window system and OSF/Motif appearance and behavior . With ESP , IVD is independent of the underlying tools.

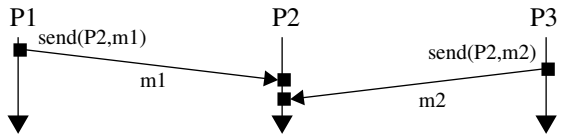
3 Program replay

3.1 What is program replay?

Most parallel programs produce deterministic output. If they are given the same input, they will produce the same output. But if one looks inside the program at the actual computations performed during different runs of the program on the same input data, the computations can be wildly different. This non-deterministic internal behavior is usually caused by race conditions that get resolved differently in different runs of the program even if the input data is the same each time. An example of a race condition is shown in Figure 4. The messages **m1** and **m2** could arrive in either order at **P2** and the order may change from one run of the program to another. This non-deterministic behavior is bad for debugging. It may not be possible to reproduce the resolution of the races that resulted in a defect when the program is re-executed using a debugger.

Program replay solves this problem through a two step process. First, the program is executed and a trace is made recording how all of the races were resolved. The program is then re-executed using the trace. The information in the trace is used to ensure that each race is resolved as it was during the initial execution when the trace was made.

FIGURE 4. Example of a race condition



This way the internal behavior of a parallel program becomes deterministic and the cyclic debugging techniques used for sequential programs can be used to debug parallel programs. To isolate any defect, the only requirement is that a trace was taken when the defect occurred.

For program replay to be completely successful, all sources (e.g. system clock, user input) of non-determinism must be traced during the initial execution and then controlled during re-execution to ensure the same behavior.

Program replay is not new. LeBlanc *et al.* [3] describe Instant Replay, a mechanism to do program replay for parallel programs in which all communication is through shared objects. Leu *et al.* [4] describe a method of doing program replay for parallel programs made up of sequential processes that communicate through message passing.

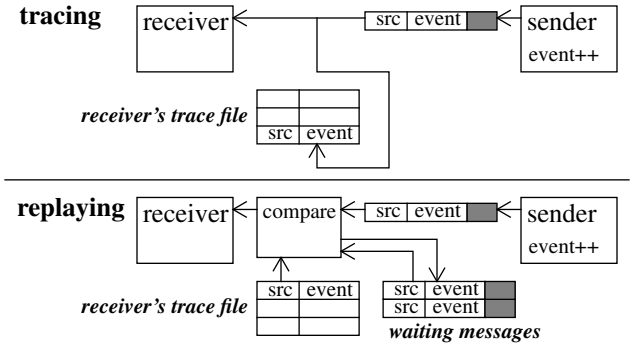
3.2 Program replay in PVM

Program replay was implemented in PVM 3.0 by modifying the PVM library, daemon, and console. One modification was to provide the functionality of program

replay . Another was to provide a user interface that allows program replay to be used with or without the rest of IVD. Details can be found in [5]. Here is an overview .

During the initial execution of a PVM program, each PVM process and daemon writes its own private trace file. In its trace file it records the order in which it received its messages. For this to be possible, each message sent during the execution of the program must have a unique identifier. To guarantee this, each message sender maintains an "event counter." Every time a message is sent, the sender increments its event counter and stores the new value into the message's header, giving the message a unique identifier made up of the sender's process id and the value of the event counter stored in the header. See **tracing** in Figure 5. The value of the event counter is shown by **event** and the sender's process id shown by **src**. When the receiver receives a message, it appends the pair <**src**, **event**> identifying the message to the end of its trace file.

FIGURE 5. How program replay works



When the program is re-executed during replay, the information in the trace files is used to force the messages to be received in the same order as they were during the initial execution. See **replaying** in Figure 5. The sender behaves as before, tagging each message that it sends with the value of its event counter. But the receiver behaves differently. This time it reads its trace file. When a message arrives, it compares the unique identifier in the message with the identifier at the head of the trace file. If they match, it advances to the next identifier in the trace file and receives the message. If they do not match, it adds the message to a pool of waiting messages. Hence, this message pool contains messages that arrived earlier during replay than they did during the initial execution. The message stays in the pool until its identifier appears at the head of the trace file. At that time, the receiver advances its trace and receives the message, removing it from the pool.

Notice that the receiver does nearly all the work of program replay, both during tracing and replay. Notice also that the trace files do not need to be merged to do replay.

It may seem that it would not be necessary to trace the PVM daemons, but this is wrong. The daemons maintain shared state that is readable and modifiable by the processes. The daemons must be traced and replayed to ensure that the processes have the same view of this state during replay.

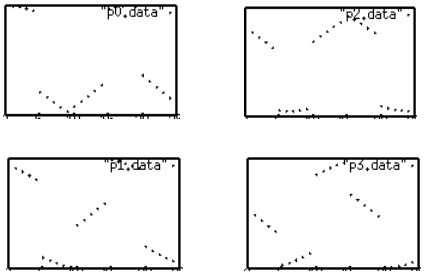
4 Data visualization

Existing debuggers are fine tools for finding errors in simple programs; they are inadequate when dealing with large, scientific codes. The problem is in looking at the data. Today's debuggers allow the user to look at the program variables one at a time or at arrays as a whole. As long as the arrays are small or the errors obvious, this view is all we need. As soon as we need to find a subtle error in an array of millions of elements, we are out of luck.

The problem is even worse for parallel programs. The current technology of debugging a distributed memory multi-computer consists of invoking an instance of the debugger for each process. If we dump the part of the array held by each process to a separate file, we can invoke a visualizer, but we may not be able to see what we need. It all depends on how the data is distributed.

For example, consider a one dimensional array of length 100 distributed over 4 nodes in blocks of 5 elements. Figure 6 shows a plot of the data held on each node. The array should look like a cosine function, but we purposely introduced a small error. It is not obvious where the error is.

FIGURE 6. A plot of the data held on each node



If the application required visualization of this array, the programmer would have written the code necessary to gather the data together and plot it. During a debugging session, though, we don't know ahead of time what arrays the user will want to see. Hence, we want an *ad hoc* scheme [6] that gathers the data without requiring the user to modify the application.

In order to put the data back together again, we need some information. First, we need to know which process produced which output file. Next, we need to know the

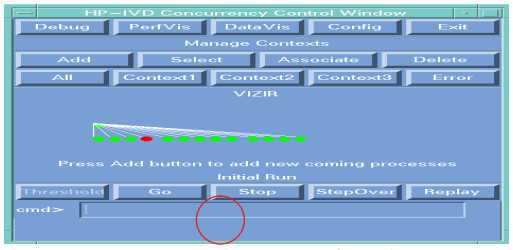
logical distribution of the array . Finally we have to know the user 's logical ordering of the processes.

Our present implementation assumes that the array is distributed in a regular grid with arbitrary blocking in each of up to 4 dimensions, allowing us to follow the time evolution of 3 dimensional objects. For each dimension, the user gives the length, the block size, and the number of processes. In addition, we allow the segments to overlap by an arbitrary amount. When the segments overlap, we accommodate toroidal distribution by allowing the last segment in each dimension to overlap the f irst. In the example we are using, there is one dimension of length 100. There are 4 processes and 5 contiguous elements are given to each process in turn, so the block size is 5.

While this simple scheme doesn' t cover all possible data distributions, most of the common ones like those found in High Performance Fortran [7] can be described. In the other cases, we allow the user to def ine an explicit data distribution which provides a lot of flexibility .

We now have all the data in files, one per process; we know which file corresponds to each entry in the process id array; and we know the data distribution. W e can now read the individual files and produce a single file with the data in the same order the debugger would have produced had we dumped it during a uniprocessor run.

FIGURE 7. The reconstructed complete array



Error found!

Figure 6 shows how difficult it can be to find a small error by looking at the pieces of the array held by each process. Figure 7 shows the reconstructed array . The data points in error are easy to spot. Simply counting the points indicates that process 2 is the culprit. W e can now go about identifying the source of the error . (In this case it is easy . We added 0.1 to all the points computed by process 2). A more sophisticated visualization could have used a different color or plot symbol for each process because the output file we create has the process id associated with each data point.

The ability to visualize the array as a whole is even more important for higher dimensional problems. For example, a 3 dimensional array might be displayed as a movie showing how 2D slices change with time.

Our tool lacks some of the features we could have added had it been fully integrated with the debugger. File I/O is slow compared to having the visualizer and debugger work together on data stored in memory. On the positive side, with IVD any debugger that can redirect its output to a file and any visualizer that can read data from a file can be used. This feature makes our tool completely portable and as flexible as most people will ever need. It also made the tool much simpler to develop.

5 Conclusions

IVD provides unique features not found in other parallel debuggers. For example, Convex's PVMdb provides a single console for controlling execution of individual existing debuggers but does not have global commands, contexts, data visualization, replay, *etc*. Also, PVMdb does not support window-based debuggers.

IVD is an on-going experiment in Hewlett-Packard Research Labs. IVD provides standard window interfaces to existing visualizers/debuggers with deterministic replay and data/performance visualization. In addition, IVD has the ability to group processes into various contexts and perform simultaneous debugging operations (*e.g.*, go, stop, single-step). Programmers may use their favorite visualizers and debuggers. Our studies have been promising. With IVD as a vehicle, we will continue to develop the overall environment shown in Figure 1.

Acknowledgment & references

Thanks to Dr. Chris Hsiung for his encouragement and suggestions.

[1] Communications of ACM, Special Section on "Graphical User Interfaces: The Next Generation", Apr. 1993, Vol. 36, No. 4.

[2] G. A. Geist *et al.*, "PVM 3.0 User's Guide and Reference Manual" ORNL/TM-12187, February, 1993.

[3] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471-82, April 1987.

[4] Eric Leu and André Schiper. Execution replay: a mechanism for integrating a visualization tool with a symbolic debugger. *Lecture Notes in Computer Science,* volume 634, Berlin, 1992.

[5] Milon Mackey, Program replay in PVM. *1993 PVM User's Group Meeting* (Knoxville, TN), May 1993. Available from netlib.

[6] Alan H. Karp, Ming C. Hao, "Ad Hoc Visualization of Distributed Arrays" HPL-93-72, 1993.

[7] David B. Loveman "High Performance Fortran", *IEEE Parallel & Distributed Technology* , 4:25-42, 1993.

On-the-Fly Visualization and Debugging of Parallel Programs

Ming C. Hao, Alan H. Karp, Milon Mackey,
Vineet Singh, Jane Chien

HPL–93?
October, 1993

Event sensing,

Parallel processing,

Program replay,

Data visualization

IVD, Interactive Visualization Debugger, is intended to provide on-line and integrated mechanisms for debugging, performance analysis, and data visualization for message-passing parallel applications. The current IVD includes: (1) ESP, a mechanism to multicast window-based commands from a single control window to some subset of existing debuggers / visualizers on various processes; (2) program replay to reproduce program runs deterministically to enable cyclic debugging; and (3) ad hoc data visualization of distributed arrays using existing visualizers.

Copyright ©1990 Hewlett-Packard Company