5 Acknowledgments

We would like to thank Robert Manchek, Keith Moore, and James Plank for their contributions to PVM, Xab, and HeNCE.

References

[1] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. A users' guide to PVM parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[2] Bernd Bruegge. A portable platform for distributed event environments. *ACM SIGPLAN Notices*, 26(12):184–193, December 1991. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.

[3] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A machine-independent communication library. In J. Gustafson, editor, *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 565–568, P.O. Box 428, Los Altos, CA, 1990. Golden Gate Enterprises.

[4] G. A. Geist and V. S. Sunderam. Experiences with network based concurrent computing on the pvm system. Technical Report ORNL/TM-11760, Oak Ridge National Laboratory, January 1991.

[5] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[6] Margaret Simmons and Rebecca Koskela. *Performance Instrumentation and Visualization.* ACM Press, Frontier Series, New York, 1990.

[7] V. S. Sunderam. PVM : A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

Cray because it is heavily loaded. The HeNCE trace tools make this kind of information intuitively obvious to the programmer. Moreover, the mapping done by HeNCE is easily adjusted via the cost matrix. The user's program need not be recompiled during this tuning process.

The HeNCE tool has evolved naturally during the course of our experience with programming various parallel machines. We are primarily motivated by the lack of uniformity and limited capabilities offered by vendors for explicit parallel programming. Our experience with HeNCE has been encouraging for the most part. We do not view it as a "solution" to the software problem we face in parallel programming. However, we do think this will be useful in the short term and perhaps will have some influence on the development of a long term solution.

3.3 Availability

An underlying philosophy of our work has been to emphasize practicality; therefore, our work is oriented towards software systems that are realizable and immediately applicable to real-life problems. The software produced is freely distributed to researchers and educators, allowing them to harness their distributed compute power into comprehensive virtual machines.

PVM and HeNCE are available by sending electronic mail to netlib@ornl.gov containing the line "send index from pvm" or "send index from hence". Instructions on how to receive the various parts of the PVM and HeNCE systems will be sent by return mail.

Xab is also available from netlib. The index from pvm explains how to obtain this software.

4 Conclusions

The focus of this work is to provide a paradigm and graphical support tools for programming a heterogeneous network of computers as a single resource. PVM and its auxiliary tools Xab and HeNCE provide the user with a set of tools to effectively use a heterogeneous network of computers in solving scientific applications.

PVM, Xab, and HeNCE are active research projects and as such are continuing to evolve and become refined. Many of the features of HeNCE will find their way into tools like Xab that eventually will fit directly into the PVM framework. These prototype systems have been built and are being used today.

There is a critical need for standards and tools to help use the high-performance computer systems available today. By building such prototype tools as outlined and using the feedback from the user community we hope to provide an easy to use, portable system for heterogeneous computing. There remains many significant research issues about this approach to parallel programming.

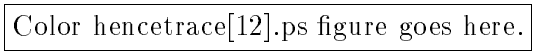

Figure 5: Tracing a parallel program in HeNCE.

icons of the active machines are illuminated with different colors depending on whether they are computing or communicating. Under each icon is a list of the node procedures mapped to this machine at any given instant. The second window displays the user's graph of the application, which changes dynamically to show the actual paths and parameters taken during a run. The nodes in the graph change colors to indicate the various activities going on in each procedure. The third window shows a histogram of the processor utilization. Figure 5 shows a snapshot of the trace tool in action.

In addition to discovering mistakes in the specification of the graph, this representation is useful in exposing more subtle aspects of the executing program, such as load balancing and network speeds. For example when the graph produced by HeNCE is contrasted with the abstract user specified graph there are noticeable differences. The graph produced by HeNCE may exposes inherent serial bottlenecks in the algorithm or a problem with various networks utilized by the computation.

Our goals here are simple: to be able to schedule and trace the flow of execution within an application and to understand where bottlenecks occur. Monitoring performance by using a timing routine has been employed in the past. This approach, however, has a number of limitations in the parallel setting. We want an animation of run-time behavior, visualizing the parallel parts in execution as the application is running. We would like to understand what performance issues arise during execution and what bottlenecks develop, and to view where programming errors cause a parallel program to get into trouble.

A main advantage of sequential debuggers is showing the point of failure. In fact many programmers only resort to using debuggers when they are mystified as to the point of failure, i.e. dividing by zero or dereferencing a null pointer. In a parallel program there may be multiple failures. Or perhaps more perplexing, one part of the program may crash while other parts continue executing for some time. One advantage of the HeNCE trace display is that it can use it's two dimensional display to inform the programmer of such problems. If part of a HeNCE program fails and other parts continue to execute, the trace tool will display the program node failure but continue to display the progress of other program nodes as they execute.

The trace animation is also important in performance tuning. Almost all the machines used with HeNCE are multitasking, this leads to unpredictable execution time profiles. The trace animation provided by HeNCE shows the programmer in real time how the program is progressing. From this animation a programmer can analyze a programs behavior and tune it to better match the execution environment. For instance, a scientist using a network of workstations and along with a Cray may realize that during certain periods of the day it is more productive to map fewer processes to the

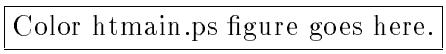

Figure 4: Composing a parallel program in the HeNCE environment.

subroutine supplied by the user. Arcs between nodes represent data dependency and control flow. A dependency arc from one node to another represents the fact that the tail node of the arc must run before the head of the arc. Data is sent to a node from its ancestors in the graph (usually its parents).

In addition to simple nodes, four types of control constructs are available in the HeNCE graph language. One represents looping; a second represents conditional dependency; a third represents a fan-out to a variable number of identical subgraphs; and a fourth represents pipelining. The graph can contain loops around subgraphs that execute a variable number of times based on the expression in the loop construct. Using a conditional construct, a section of the graph can be executed or bypassed based on an expression that will be evaluated at run time. A variable fan-out (and subsequent fan-in) construct is available while composing the graph. The width of the fan-out is specified as an expression that is evaluated at run time. This construct is similar to a parallel-do construct found in several parallel Fortrans. In pipelined sections, when a node finishes with one set of input data, it reruns with the next piece of pipelined data.

Once the dynamic graph is specified, a **configuration** tool in the HeNCE environment can be used to specify the configuration of machines that will compose his parallel virtual machine. The configuration tool also assists the user in setting up a cost matrix. The cost matrix allows the user to describe which machine can perform which task and can give priority to certain machines. HeNCE will use this cost matrix at run time to determine the most effective machine on which to execute a particular procedure in the graph.

The HeNCE environment also contains a **build** tool to perform three tasks. First, by analyzing the graph, HeNCE automatically generates the parallel program using PVM calls for all the communication and synchronization required by the application. Second, by knowing the desired PVM configuration, HeNCE automatically compiles the node procedures for the various heterogeneous architectures. Finally, the build tool installs the executable modules on the particular machines in the PVM configuration.

The **execute** tool in the HeNCE environment starts up the requested virtual machine and begins execution of the application. During execution, HeNCE automatically maps procedures to machines in the heterogeneous network based on the cost matrix and the HeNCE graph. Trace and scheduling information that is saved during the execution can be displayed in real time or replayed later.

The HeNCE environment has a **trace** tool that allows visualization of the parallel run. The trace tool is X-window based and consists of three windows. One window shows a representation of the network and machines underlying PVM. In this window

3.2 HeNCE

In developing software, the initial definitions and specifications are often performed graphically; flow charts or dependency graphs are well known examples. The overall structure of the problem can be visualized far more easily from these graphical representations than from textual specifications; from the development point of view, such a representation enhances the quality of the resulting software. However, in order to be executed, such a description must be converted to program form, typically manifested as source code. These graphical representations therefore, must eventually be translated to operational programs — the graphical depiction of a concurrent application, and strategies for its successful execution on a heterogeneous network are the two fundamental inputs to the HeNCE environment.

With the use of a graphics interface, implemented on a workstation for example, a user can develop the parallel program as a computational graph, where the nodes in the graph represent the computation to be performed and the arcs represent the dependencies between the computations. From this graphical representation, a lower-level portable program can be generated, which when executed will perform the computations specified by the graph in an order consistent with the dependencies specified. This programming environment allows for a high-level description of the parallel algorithm and, when the high-level description is translated into a common programming language, permits portable program execution. This environment presents the algorithm developer with an abstract model of computation that can bind effectively to a wide variety of existing parallel processors. Specific machine intrinsics may be confined to the internal workings of such a tool in order to provide a common user interface to these parallel processors.

Another problem facing the developers of algorithms and software for parallel computers is the analysis of the performance of the resulting programs. Often performance bugs are far more difficult to detect and overcome than the synchronization and data dependency bugs normally associated with parallel programs. We have developed a fairly sophisticated postprocessing performance analysis tool associated with the graphics programming interface just described. This tool is quite useful in understanding the flow of execution and processor utilization within a parallel program.

HeNCE provides the programmer with a higher level environment for using heterogeneous networks. The HeNCE philosophy of parallel programming is to have the programmer explicitly specify the parallelism of a computation and to automate, as much as possible, the tasks of writing, compiling, executing, debugging, and analyzing the parallel computation. Central to HeNCE is an X-Window interface that the programmer uses to perform these functions (see Figure 4).

The HeNCE environment contains a **compose** tool that allows the user to explicitly specify parallelism by drawing a graph of the parallel application. (If an X-window interface is not available, then textual graph descriptions can be input.)

Each node in a HeNCE graph represents a procedure written in either Fortran or C. The procedure can be a subroutine from an established library or a special purpose

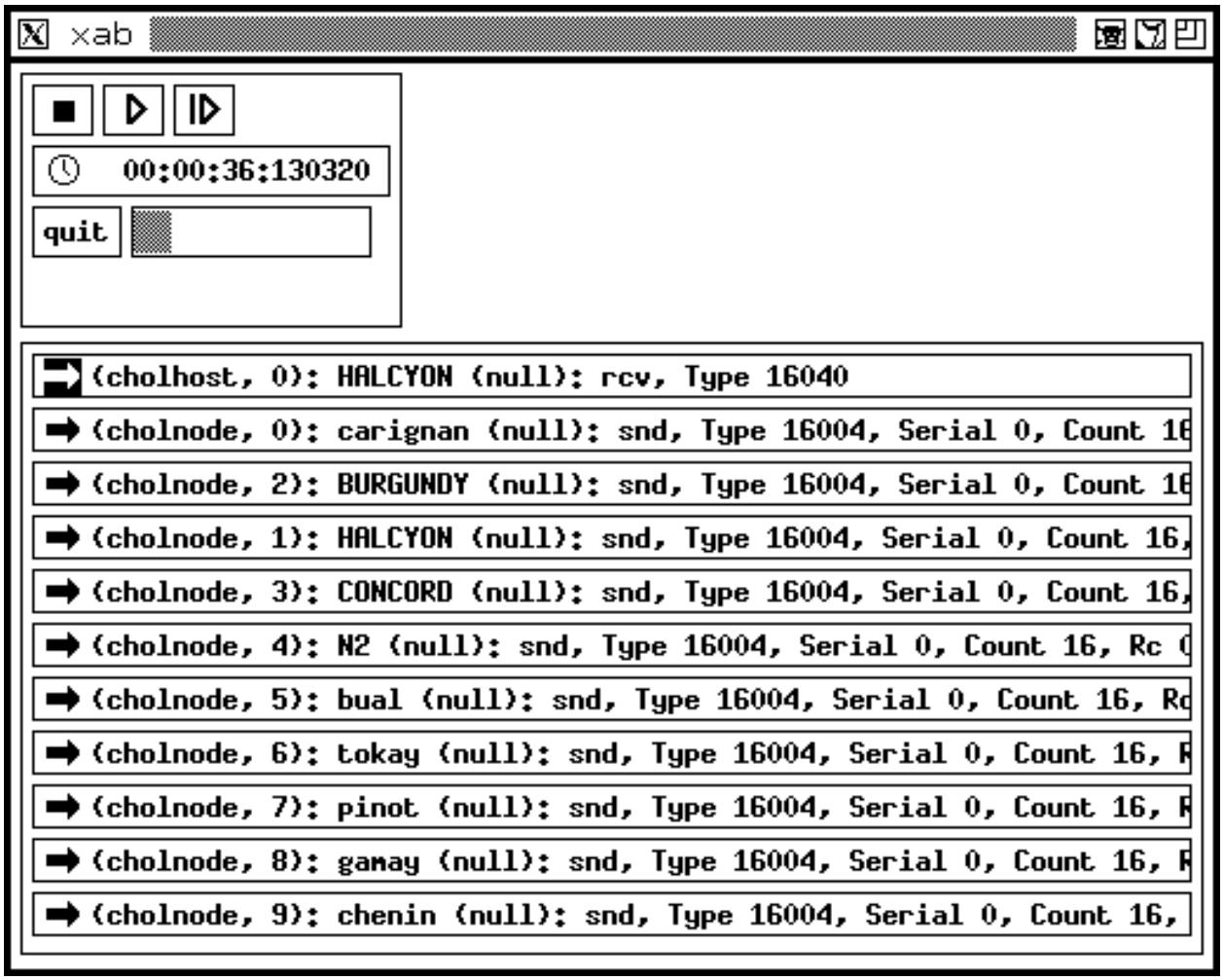

Figure 2: Xab showing a deadlocked version of the Cholesky example.

Color pgchol.ps figure goes here.

Figure 3: Postmortem ParaGraph views of the Cholesky program.

as n grows. In fact, when n reaches the number of events in the program, the real time monitor becomes postmortem. As the display lags behind the program state, some problems in program behavior cannot be detected. The example in this section illustrates this point. Another factor one must consider is the amount of memory required to store events before sending them to the monitor. With Xab, events are immediately dispatched to the monitor. As a result, Xab adds little, in terms of memory requirements, to the PVM processes being monitored.

An Example

One of the example programs that comes with PVM is a distributed matrix decomposition program based on a Cholesky factorization of the matrix. The window in Figure 1 is the Xab display in progress for this program. The host process, `(cholhost, 0)` is blocked on a receive. Process `(cholnode, 0)` has just received a message. The node process `(cholnode, 1)` is extracting data from a message buffer while `(cholnode, 2)` is placing eight floats into a message buffer.

One of the advantages of the real time display of Xab is its ability to immediately show events as they happen. For instance, Figure 2 shows the same Cholesky example but this time an error has been deliberately introduced. The host is waiting on a message of type 16040 while all the cholnode processes are sending messages of type 16004, thus the program has deadlocked. This example illustrates the advantage of real time monitoring. Postmortem monitoring would not work in this situation since the program would not complete and therefore would never flush the events for display.

There are several research projects which focus on displaying events generated by distributed memory parallel programs, notably ParaGraph [5] and Bee [2]. Currently Xab events are stored in Xab's own ascii based format. Due to ParaGraph's wide availability, we provide an awk script called `topg`, which converts Xab event files to the PICL format [3] so they can be used in ParaGraph. The ParaGraph tool provides a rich set of displays for visualizing message passing parallel programs. Figure 3 shows a ParaGraph visualization of the PVM Cholesky program with one host process and two slave processes. There are several differences between the ParaGraph display and the Xab display. ParaGraph provides a variety of views but is limited to postmortem visualization. Xab currently has a limited display facility but can operate in real time. The ParaGraph trace events need to be in temporal order. Xab simply displays events as they arrive. ParaGraph was developed originally for multicomputers which did not support multitasking. As a result, ParaGraph nodes are the same as processes. PVM programs name processes by a component name and an integer pid. In order to use ParaGraph for the display of Xab traces, the component/pid pairs are mapped to an integer node number which may not be intuitive.

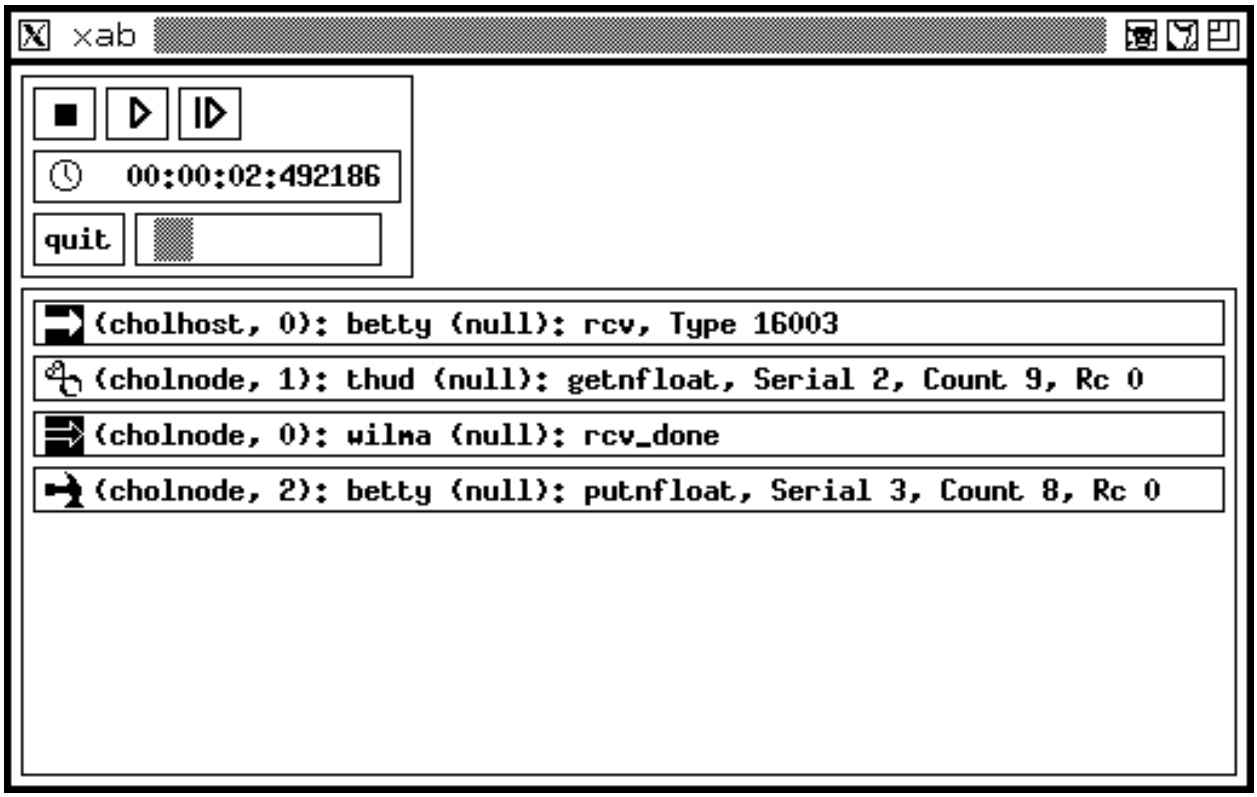

Figure 1: Xab display while monitoring the PVM Cholesky demo.

event generated at the start of a barrier, it is the name of the barrier and the number of processes that must reach the barrier before continuing. Other event messages contain similar event specific information.

Besides the event messages, Xab also inserts one additional piece of information into user messages. Each message sent by the user program is given a serial number. This serial number is prepended to the user's message buffer at `initsend()` time and stripped from the buffer at receive time. The addition of this single integer to user messages facilitates analysis of messages, since every message can be uniquely identified by its source process and serial number.

Monitoring processes

The `abmon` process receives event messages from the instrumented PVM calls and formats them into human readable form. The `abmon` program must be running before the user's program starts since it needs to receive event messages from the instrumented calls. The formatted event messages can either be written to a file or sent to the Xab display program. Just as an astronomer on Earth observes events that have traveled various distances, the `abmon` process observes events relative to its position in the virtual machine. When `abmon` formats events it also adds its own perspective within the virtual machine by placing local time stamps into the event record. The additional time stamps may be used to discern `abmon`'s perspective by indicating how long it takes events to propagate from a user process to the monitor process.

The display process will take events as formatted by `abmon` and display them in a window as shown in Figure 1. Xab supports two modes of event playback, continuous play or single step. When the play button is pressed the events will be displayed in real time. The slider controls the speed of the playback in continuous playback mode. Playback may be stopped at any time by pressing the stop button. The single step button will show only the next event.

The following command line executes Xab, displaying the events in real time and saving them in a file for later review:

```
% abmon | tee evfile | xab
```

The `abmon` program reads event messages and writes them to standard out. The UNIX command `tee` copies the events to the file `evfile` and also passes them to `xab` via the pipe. The `xab` program actually opens a window and displays the events.

Timeliness versus message traffic

The method described previously for sending Xab monitor messages is utilized for every user call to the PVM library. This approach generates what may be considered an inordinately large number of messages. There is a tradeoff between the number of messages and the timelyness of the event display. If events are buffered and sent to the monitor after every n events then the event display becomes more asynchronous

Event messages

The Xab libraries call the normal PVM functions for the user but they also send PVM messages to a special monitoring process, called `abmon`. Covertly sending PVM messages is tricky business. Care must be taken not to disrupt any PVM messages the user's program may be manipulating. The current version of PVM (2.4.1 at the time of this writing) supports a single message buffer. In other words, a PVM message is built by constructing a buffer and then sending it off to one or more locations. A typical sequence of commands is:

```
initsend();       /* start a new message buffer  */
putnint(&i, 1);   /* place an int in the buffer  */
putnfloat(&f, 1); /* place a float in the buffer */
snd("compnode", 1, 63);  /* send the buffer too (compnode,1) */
snd("comppart", 4, 63);  /* and too (comppart,4) as msgtype 63   */
```

Here the message buffer is initialized via `initsend()`. An integer and a float are placed into the buffer. Finally this buffer is sent to two different PVM processes. (Processes in PVM are addressed by a name and an instance number.) When the program is instrumented, each of these lines of code generates a message to the abmon monitoring process. The Xab message is itself a PVM message. Precautions must be taken so the Xab message does not destroy the user's message. Xab takes the liberty of altering internal PVM data structures in a very limited way. Before Xab sends one of its event messages, it stores PVM's internal pointers to the user's message and reinitializes these pointers so the Xab message can be built. After the Xab message is sent, the original internal PVM pointers are restored, thus reinstating the user's message buffer. Since Xab only sends covert messages and never receives them, PVM's internal data structures for received messages do not need to be altered. Multiple message buffers will be supported in the next PVM release so that even this small altering will be unnecessary.

The content of the Xab event messages generally include an event type, a time stamp (in microseconds), and event specific information. The event type indicates which PVM call is being invoked. In some cases a PVM call may generate two events. For instance, the PVM barrier function generates an event before and after the barrier call. This allows the user to see when barriers are initiated as well as completed. The time stamp in the event message is the time of day on the machine where the PVM call is being executed. It is possible that the clocks on various machines involved in a computation will not be synchronized. Xab does not rely on the clocks being synchronized; events are simply displayed as they arrive. Although future versions of Xab may make use of the time stamps it is not always necessary that the machine clocks be synchronized. For instance, it may be informative to know how long processes wait at a particular barrier in a program. Xab could use the time stamps from barrier events to display this information, independent of relative machine clock synchronization. The event specific information in an Xab message varies for different PVM routines. For the

3 Monitoring, Debugging, and Performance Tuning

It is important that tools assist the programmer in the writing, debugging, and performance tuning of applications. It is not uncommon for concurrent application performance to improve by an order of magnitude, as a result of small-scale changes based on an analysis of execution profiles, communication patterns, and load imbalances. Previous research in visualization has focused on homogeneous parallel processing for both shared and distributed memory machines [6]. Our work focuses on visualization and debugging for networks of heterogeneous computers. We have developed a set of tools to help users with the complex task of understanding a program's behavior both from a correctness and a performance standpoint. In what follows we describe two systems, Xab and HeNCE, which are built on top of PVM and are intended to help users with these tasks.

3.1 Xab

While PVM provides a solid programming base, it does not provide the user with many options for analyzing or debugging PVM programs. For this reason Xab was developed to aid the user in the development of PVM programs. Xab is a run-time monitoring tool for PVM programs that gives the user direct feedback as to what PVM functions his or her program is performing. Xab is composed of three parts: an Xab library to which the user links applications, a PVM process called `abmon` that quietly receives tracing messages from the library routines, and a display process called `xab` that is a graphical X-Window display of the trace events.

The approach of real time monitoring is particularly apropos in a heterogeneous multiprogramming environment where differences in computation and communication speeds are due both to heterogeneity and external CPU and network loads. Monitoring can help give the user insight into how a program is behaving in such an environment.

Xab monitors a PVM program by instrumenting calls to the PVM library. The instrumented calls generate events that can be displayed during program execution. The PVM calls are instrumented by replacing them with calls to the equivalent Xab library routines.

A Fortran program normally accesses the PVM user routines via the f2c library that comes with PVM. With Xab, the f2x library replaces the f2c library. This allows Fortran programs to use Xab simply by linking to libf2x in place of libf2c. With C, the procedure is slightly more complicated. The programmer must add the include file xab.h to source files that call PVM routines and then recompile the modified source files. This include file contains macros that replace the normal PVM routines with calls to the Xab library. Both Fortran and C programs must be linked with the Xab library, called libab.

SPARCstation, Microvax, DECstation, IBM RS/6000, HP-9000, Silicon Graphics IRIS, NeXT, Sequent Symmetry, Alliant FX, IBM 3090, Intel iPSC/860, Thinking Machines CM-2 and CM-5, KSR-1, Convex, Cray Y-MP, and Fujitsu VP-2000. In addition, users can port PVM to new architectures by simply modifying a generic 'makefile' supplied with the source and recompiling.

Using PVM, each user can configure his own parallel virtual computer, which can overlap with other users' virtual computers. Configuring a personal parallel virtual computer involves simply listing the names of the machines in a file that is read when PVM is started. Several different physical networks can co-exist inside a virtual machine. For example, a local ethernet, HIPPI, and a fiber optic network can all be a part of a user's virtual machine. While each user can have only one virtual machine active at a time, PVM is multitasking so several applications can run simultaneously on a parallel virtual machine.

The PVM package is small (less than 1 MByte) and easy to install. It needs to be installed only once on each machine to be accessible to all users. Moreover, the installation does not require special privileges on any of the machines and thus can be done by any user.

Application programs that use PVM are composed of subtasks at a moderately large level of granularity. The subtasks can be generic serial codes, or they can be specific to a particular machine. In PVM, compute resources may be accessed at three different levels: the *transparent* mode in which subtasks are automatically located at the most appropriate sites, the *architecture-dependent* mode in which the user may indicate specific architectures on which particular subtasks are to execute, and the *machine-specific* mode in which a particular machine may be specified. Such flexibility allows different subtasks of a heterogeneous application to exploit particular strengths of individual machines on the network.

The PVM programming interface requires that all message data be explicitly typed. PVM performs machine-independent data conversions when required, thus allowing machines with different integer and floating point representations to pass data. Applications access PVM resources via a library of standard interface routines. These routines allow the initiation and termination of processes across the network as well as communication and synchronization between processes. Communication constructs include those for the exchange of data structures as well as high-level primitives such as broadcast, barrier synchronization, and rendezvous.

Application programs under PVM may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, the processes in existence may have arbitrary relationships between each other and, further, any process may communicate and/or synchronize with any other.

Heterogeneous networks of computers are becoming commonplace in high-performance computing. Systems ranging from workstations to supercomputers are linked together by high speed networks. Until recently each computing resource on the network remained a separate unit, but now over 100 institutions worldwide are using the PVM software package to develop truly *heterogeneous* programs utilizing multiple computer systems to solve applications. PVM stands for Parallel Virtual Machine [7,4,1]. PVM was designed with heterogeneity and portability as primary goals. As such it allows machines with different architectures and floating point representations to work together on a single computational task.

In the development of heterogeneous concurrent applications for heterogeneous target environments, large grained sub-task partitioning and processor allocation are critical issues. Additionally, the construction of program modules, specification of interdependencies and synchronization, and the management of multiple objects for different architectures are tedious activities that are prone to errors. To address these issues and to provide at least partial solutions, we have developed two packages on top of PVM to aid in the use, programming, and analysis of parallel computers. These packages are called Xab and HeNCE.

Xab (X-window Analysis and deBugging) is a tool for run-time monitoring of PVM programs. Using Xab, PVM programs can easily be instrumented and monitored by simply relinking to the Xab libraries. Xab is itself a PVM program. This makes Xab very portable but it leads to interesting issues of how to make Xab peacefully coincide with the programs it monitors.

HeNCE (Heterogeneous Network Computing Environment) is an environment that is concerned with the development of high level techniques for programming concurrent virtual machines of the type provided by PVM. The goal of HeNCE is to simplify the task (and thus reduce the chance of error) of programming a heterogeneous network of computers while still providing the programmer with access to the high performance available from such configurations.

2 PVM

PVM is a software package that permits a heterogeneous collection of serial, parallel and vector computers hooked together by a network to appear as one large computer. Thus, PVM allows a user to exploit the aggregate power of workstations and supercomputers distributed around the world to solve computational grand challenges.

The user views PVM as a loosely coupled distributed-memory computer programmed in C or Fortran with message-passing extensions. The hardware that composes the user's personal PVM may be any UNIX based machine on which the user has a valid login and is accessible over some network.

PVM may be configured to contain various machine architectures including sequential processors, vector processors, multicomputers, etc. The present version of the software has been tested with various combinations of the following machines: Sun3,

It is recognized that special views are required to understand the complex interactions between tasks in a heterogeneous environment. Problems of network delays and dynamically changing machine loads are important considerations in trying to debug or improve the performance of a PVM application. From a different perspective, certain types of monitoring information can be valuable as application debugging aids, particularly when transformed appropriately or augmented with code-specific information. We present several ideas for such visual displays, and methods for debugging programs in a heterogeneous environment.

Keywords: parallel algorithms, graphics tools, visualization, programming environments, performance studies, performance tuning, portability, heterogeneous network computing

1 Introduction

The emergence of a wide variety of commercially available parallel computers has created a software dilemma. Will it be possible to design general-purpose software that is both efficient and portable across a wide variety of these new parallel computers? Moreover, will it be possible to provide programming environments that will be sophisticated enough to make explicit parallel programming a viable means to exploit the performance of these new machines? For many computational problems, the design, implementation, and understanding of efficient parallel algorithms can be a formidable challenge. Efficient parallel programs are more difficult to write and understand than efficient sequential programs, because there are additional issues of synchronization and coordination of multiple tasks. Parallel programs are often less portable than serial codes, because their structure may depend critically on specific architectural features of the underlying hardware (such as the way in which data sharing, memory access, synchronization, and process creation are handled).

The computing requirements of many current and future applications, ranging from scientific computational problems in the material and physical sciences, to simulation, engineering design, and circuit analysis, are best served by concurrent processing. While hardware multiprocessors can frequently address the computational requirements of these high-performance applications, there are a number of other aspects to concurrent computing that are not adequately addressed when conventional parallel processors are used to solve these problems.

For instance, software aspects, including program development methods, scalable programs, profiling tools, and support systems are among the areas of concurrent computing in which significant development is required. While hardware and architectural advances in parallelism have been rapid, the software infrastructure has not kept pace, resulting in unsystematic and ad-hoc approaches to the implementation of concurrent applications. In recent years, several research groups have focussed on various aspects of this shortcoming, and this effort has resulted in significant developments in programming paradigms, data partitioning, algorithms, languages, and scheduling.

Visualization and Debugging in a Heterogeneous Environment *

Adam Beguelin
Carnegie Mellon University and Pittsburgh Supercomputer Center

Jack Dongarra
University of Tennessee and Oak Ridge National Laboratory

Al Geist
Oak Ridge National Laboratory

Vaidy Sunderam
Emory University

Abstract

The PVM system is gaining widespread acceptance as a methodology and toolkit for heterogeneous distributed computing. With such an increase in popularity and adoption, the need for auxiliary support mechanisms and tools becomes critical. This paper describes our experiences with visualization and debugging of PVM applications. It also describes a graphical interface to programming a heterogeneous network of computers built on PVM called HeNCE. The paper includes our early work in visualization and debugging, the present research, and the future directions of our research in this area.

One of our research projects revolves around the development of a graphical monitoring package called Xab. The basic concept is to instrument every PVM call so the progress of the parallel program can be displayed in real time or saved in a trace file for later analysis. Xab can convert its trace files to PICL format so that several other graphical tracing packages such as ParaGraph can be used for detailed analysis.

*This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract DE-AC05-84OR21400, and in part by the the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.