Visualization and Debugging in a Heterogeneous Environment

Adam Beguelin, Carnegie Mellon University and Pittsburgh Supercomputing Center

Jack Dongarra, University of Tennessee and Oak Ridge National Laboratory

Al Geist, Oak Ridge National Laboratory

Vaidy Sunderam, Emory University

T he 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 these new parallel computers? Moreover, will it be possible to provide programming environments sophisticated enough for explicit parallel programming 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. Additional issues of synchronization and multiple-task coordination make efficient parallel programs more difficult to write and understand than efficient sequential programs. Parallel programs are often less portable than serial codes because their structure may depend critically on the hardware's specific architectural features (such as how it handles data sharing, memory access, synchronization, and process creation).

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. Multiprocessors can frequently address the computational requirements of these high-performance applications, but other aspects of concurrent computing are not adequately addressed when conventional parallel processors are used.

For instance, software aspects, including program development methods, scalable programs, profiling tools, and support systems, require significant development. 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 focused on various aspects of this shortcoming, producing significant developments in programming paradigms, data partitioning, algorithms, languages, and scheduling.

Heterogeneous networks of computers ranging from workstations to supercomputers are becoming commonplace in high-performance computing. Until recently, each computing resource on the network remained a separate unit, but now hundreds of institutions worldwide are using the Parallel Virtual Machine[1] soft-

---

A monitoring tool and a graphical interface working on top of the PVM software can help programmers make better use of heterogeneous networks of computers.

PVM: Heterogeneous distributed computing

PVM (Parallel Virtual Machine) is a software package being developed by Oak Ridge National Laboratory, the University of Tennessee, and Emory University. It enables a heterogeneous collection of Unix computers linked by a network to function as a single large parallel computer. Thus, large computational problems can be solved by the aggregate power and memory of many computers.

PVM supplies the functions to start tasks and lets the computers communicate and synchronize with each other. It survives the failure of one or more connected computers and supplies functions for users to make their applications fault tolerant. Users can write applications in Fortran or C and parallelize them by calling simple PVM message-passing routines such as pvm_send() and pvm_recv(). By sending and receiving messages, application subtasks can cooperate to solve a problem in parallel.

PVM lets subtasks exploit the type of computer best suited for finding their solution. Thus some subtasks may run on a vector supercomputer and others on a parallel computer or powerful workstation. PVM applications can be run transparently across a wide variety of architectures; PVM automatically handles all message conversion required if linked computers use different data representations. Participating computers can be distributed anywhere in the world and linked by a variety of networks.

The PVM source code and user's guide are available by electronic mail. The software is easy to install. The source has been tested on Sun, DEC, IBM, HP, Silicon Graphics Iris, Data General, and Next workstations, as well as parallel computers by Sequent, Alliant, Intel, Thinking Machines, BBN, Cray, Convex, IBM, and KSR. In addition, Cray Research, Convex, IBM, Silicon Graphics, and DEC supply and support PVM software optimized for their systems.

PVM is an enabling technology. Hundreds of sites around the world already use PVM as a cost-effective way to solve important scientific, industrial, and medical problems. PVM users include petroleum, aerospace, chemical, pharmaceutical, computer, medical, automotive, and environmental cleanup companies. Department of Energy and NASA laboratories use PVM for research, and numerous universities around the US use it for both research and teaching.

The software described in this article is freely distributed to researchers and educators, allowing them to harness their distributed computation power into comprehensive virtual machines. PVM and Hence are available by sending electronic mail to netlib@ornl.gov containing the line "send index from pvm3" 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. PVM problems or questions can be sent to pvm@msr.epm.ornl.gov for a quick and friendly reply.

---

ware package to develop truly heterogeneous programs utilizing multiple computer systems to solve applications (see sidebar). We designed PVM with heterogeneity and portability as primary goals. It lets machines with different architectures and floating-point representations work together on a single computational task.

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

Xab (X Window Analysis and Debugging) is a tool for runtime monitoring of PVM programs. Using Xab, programmers can easily instrument and monitor PVM programs by simply relinking to the Xab libraries. Xab is itself a PVM program, so it is very portable. However, making it peacefully coincide with the programs it monitors is problematic.

Hence (Heterogeneous Network Computing Environment) is an environment for the development of high-level programming techniques for the type of concurrent virtual machines provided by PVM. Its goal 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. There are several systems with goals similar to those of Hence. The Code system[2] and Paralex[3] both allow graph-based high-level specifications of parallel programs. Code includes tools that can map the specification into several different parallel languages or libraries such as Ada or C with shared-memory extensions. Paralex directly maps its specifications into C with calls to the Isis library.[4]

PVM

With PVM, users can exploit the aggregate power of distributed workstations and supercomputers to solve the computational Grand Challenges.

Users view PVM as a loosely coupled distributed-memory computer programmed in C or Fortran with message-passing extensions. The hardware that constitutes a user's personal PVM may be any Unix-based network-accessible machine on which the user has a valid login.

We have tested the software with combinations of the following machines: Sun3, 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, Fujitsu VP-2000, DEC Alpha, Intel Paragon, and Cray C90. In addition, users can port PVM to new architectures by simply modifying a generic "makefile" supplied with the source and recompiling.

Using PVM, users can configure their own parallel virtual computers, 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 coexist inside a virtual machine. For example, a local Ethernet, a Hippi (High-Performance Parallel Interface), and a
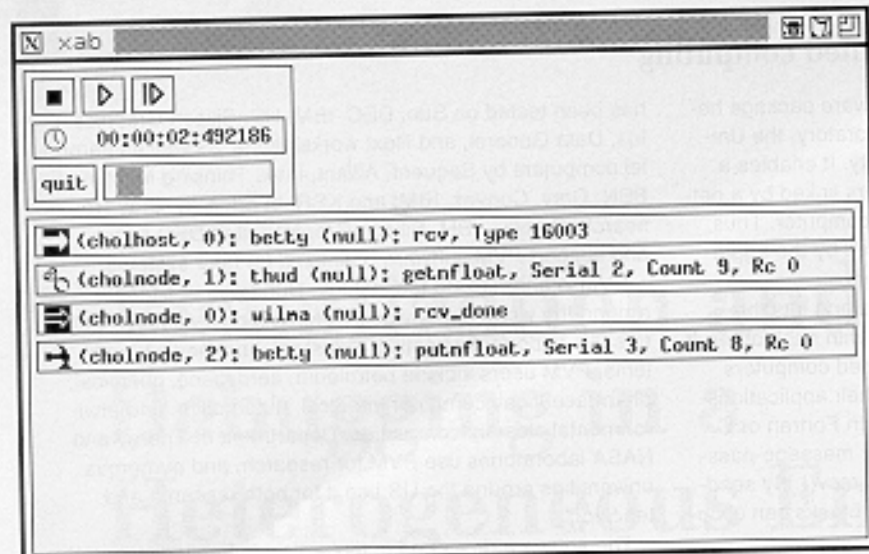
```
┌─────────────────────────────────────────────────────┐
│ [N] xab                                    [▣][▢][▣] │
├─────────────────────────────────────────────────────┤
│ ┌──────────────────┐                                 │
│ │ ■   ▷   ▮▷        │                                 │
│ │ ⊙   00:00:02:492186                                │
│ │ ┌────┐ ┌──────┐   │                                 │
│ │ │quit│ │▓▓    │   │                                 │
│ └──┴────┴─┴──────┘                                   │
│ ┌─────────────────────────────────────────────────┐ │
│ │⊟▷ (cholhost, 0): betty (null): rcv, Type 16003  │ │
│ ├─────────────────────────────────────────────────┤ │
│ │⊕▷ (cholnode, 1): thud (null): getnfloat, Serial 2, Count 9, Rc 0│ │
│ ├─────────────────────────────────────────────────┤ │
│ │⊟▷ (cholnode, 0): wilma (null): rcv_done         │ │
│ ├─────────────────────────────────────────────────┤ │
│ │─▷ (cholnode, 2): betty (null): putnfloat, Serial 3, Count 8, Rc 0│ │
│ └─────────────────────────────────────────────────┘ │
└─────────────────────────────────────────────────────┘
```

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

fiber-optic network can all be part of a user's virtual machine. Each user can have only one virtual machine active at a time; however, since PVM is multitasking, several applications can run simultaneously on a parallel virtual machine.

The PVM package is small (approximately 1 Mbyte) and easy to install. It needs to be installed only once on each machine to be accessible to all users. Moreover, installation does not require special privileges on any machines, so any user can do it.

Application programs that use PVM are composed of subtasks at a moderately coarse level of granularity. The subtasks can be generic serial codes or specific to a particular machine. In PVM, the user may access computational resources 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 can indicate specific architectures on which particular subtasks are to execute, and
- the *machine-specific* mode, in which the user can specify a particular machine.

Such flexibility lets different subtasks of a heterogeneous application exploit particular strengths of individual machines on the network.

The PVM programming interface requires that programmers explicitly type all message data. PVM performs machine-independent data conversions when required, thus letting machines with different integer and floating-point representations 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 among processes.

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

While PVM is a very popular system for programming heterogeneous networks of computers, it is not the only system of this type. The p4 system[5] from Argonne National Laboratory, Express[6] from Parasoft, and Linda[7] from Scientific Computing Associates provide functionality similar to that of PVM.

Monitoring, debugging, and performance tuning

Tools should help programmers write and debug applications and tune their performance. With small-scale changes based on analysis of execution profiles, communication patterns, and load imbalances, programmers can improve concurrent application performance by an order of magnitude. Previous research in visualization focused on homogeneous parallel processing for both shared- and distributed-memory machines.[8] Our work focuses on visualization and debugging for networks of heterogeneous computers. Xab and Hence provide tools

to help users with the complex task of understanding a program's behavior for both correctness and performance.

Xab. While PVM provides a solid programming base, it does not give users many options for analyzing or debugging PVM programs. To help in the development of PVM programs, Xab, a runtime monitoring tool, gives users direct feedback about the PVM functions their programs are performing. Xab has 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.

Real-time monitoring is particularly apropos in a heterogeneous multiprogramming environment where differences in computation and communication speeds result from both heterogeneity and external CPU and network loads. Monitoring gives the user insight into program behavior in such an environment.

Xab monitors a PVM program by instrumenting calls to the PVM library. The instrumented calls generate events displayed during program execution.

A Fortran program normally accesses the PVM user routines via the libfpvm library that comes with PVM. Fortran programs use Xab by simply linking to libxpvm in place of libfpvm. 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*.

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*.

Xab event messages generally contain 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 lets the user see when barriers are initiated and

completed. The time stamp in the event message is the time of day on the machine executing the PVM call. The clocks on various machines involved in a computation may not be synchronized, so Xab does not rely on synchronized clocks. Events are simply displayed as they arrive.

Although future versions of Xab may use the time stamps, it is not always necessary to synchronize machine clocks. For instance, it may be informative to know how long processes wait at a particular program barrier. 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 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 is given a serial number, prepended to the user's message buffer so that every message can be uniquely identified by its source process and serial number. Currently, we are exploring the usefulness of adding pseudo time stamps to Xab. Pseudo time stamps combine real clocks and logical clocks.

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 be either 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. To discern its perspective, abmon may use the additional time stamps to ascertain how long it takes events to propagate from a user process to the monitor process.

The display process takes events formatted by abmon and displays them in

a window, as shown in Figure 1. Xab supports two modes of event playback: continuous play or single step. When the user presses the play button, the events are displayed in real time. The slider controls the playback speed in continuous-play mode. Users can stop playback 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 output. 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. Every user call to the PVM library uses the method for sending Xab monitor messages described in the previous section. This approach generates an inordinately large number of messages. There is a trade-off between the number of messages and the timeliness of the event display. If events are buffered and sent to the monitor after every n events, then the event display becomes more asynchronous as n grows. In fact,

when n reaches the number of events in the program, the monitor provides postmortem rather than real-time information. Since the display lags behind the program state, users cannot detect certain problems in program behavior. (We give an example in the next section.) Another factor that must be considered is the memory required to store events before sending them to the monitor. Xab immediately dispatches events. As a result, it adds little, in terms of memory requirements, to the PVM processes it monitors. We are exploring the possibility of allowing the user to dynamically alter the event flow. This extension requires the addition of bidirectional data exchanges to the one-way dataflow currently used for Xab's monitoring.

An example. An example program that comes with PVM 2.4 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.

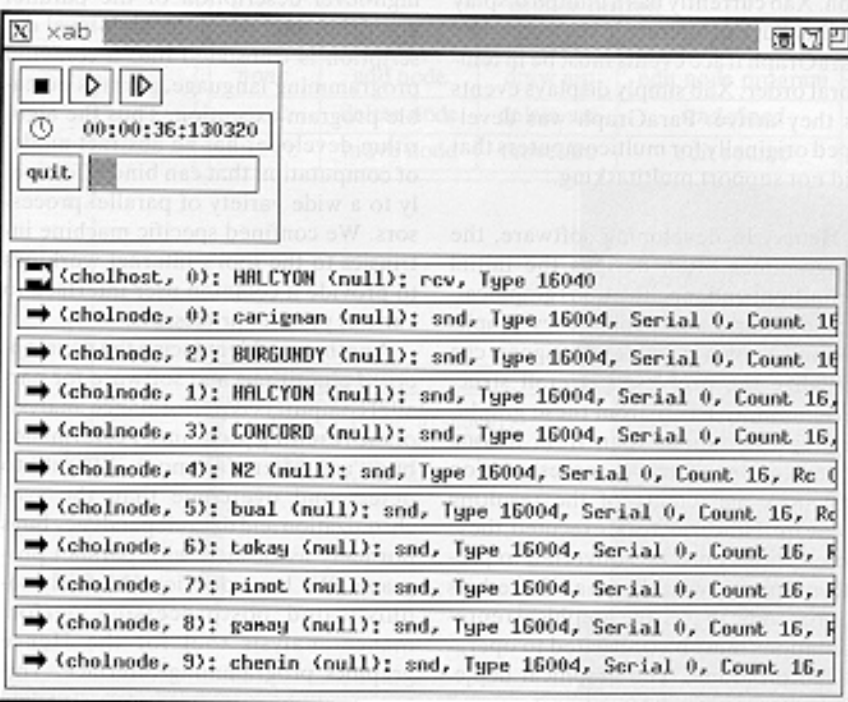As shown in Figure 2, an advantage of Xab's real-time display is its ability to

Figure 2. Xab displays an error in a Cholesky program.

detect errors in a dynamic environment. The same Cholesky example contains a deliberately introduced error. The host is waiting for a message of type 16040, while all the cholnode processes are sending messages of type 16004; thus the program has blocked indefinitely. In this case, postmortem monitoring would not work: The program would not complete and therefore would never flush the events for display.

Several research projects focus on displaying events generated by distributed-memory parallel programs, notably ParaGraph,[9] Pablo,[10] Upshot,[11] and Bee.[12] Currently, Xab events are stored in Xab's own ASCII-based format. Because of ParaGraph's wide availability, we provide a program that converts Xab event files to a ParaGraph-compatible format. 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 and the Xab displays. 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 must be in temporal order. Xab simply displays events as they arrive. ParaGraph was developed originally for multicomputers that did not support multitasking.

Hence. In developing software, the programmer often designs the initial definitions and specifications graphically; flowcharts and dependency graphs are well-known examples. Designers can visualize the problem's overall structure far more easily from these graphical representations than from textual specifications. Such a representation enhances the quality of the resulting software. However, to be executed, these descriptions must be converted to program form, typically manifested as source code; that is, the graphical representations must be translated to operational programs. The graphical depiction of a concurrent application and strategies for its successful execution
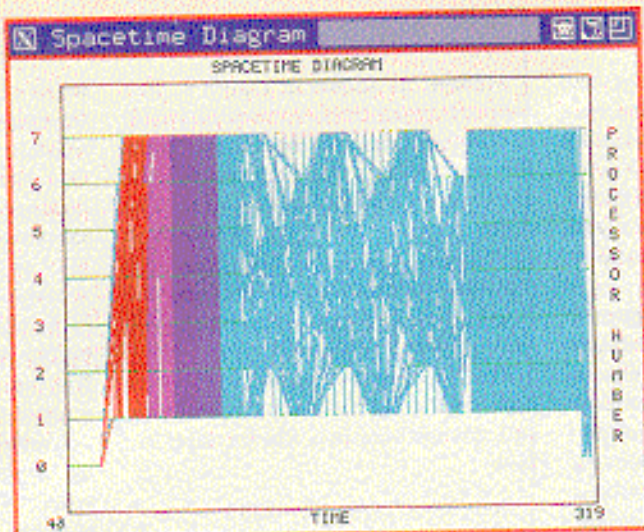

Figure 3. ParaGraph views of the Cholesky program.

on a heterogeneous network are the two fundamental inputs to the Hence environment.

With the Hence graphics interface implemented on a workstation, a user can develop a parallel program as a computational graph: the nodes in the graph represent the computations to be performed and the arcs represent the dependencies between the computations. From this graphical representation, Hence can generate a lower level portable program, 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. Thus the algorithm developer has an abstract model of computation that can bind effectively to a wide variety of parallel processors. We confined specific machine intrinsics to the tool's internal workings to provide a common user interface to various parallel processors.

Another problem facing the developers of algorithms and software for parallel computers is performance analysis of the resulting programs. Performance bugs are often 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 for the Hence graphics programming interface. This tool is quite useful in understanding the execution flow and processor utilization in a parallel program.

Hence gives the programmer 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 lets the user explicitly specify parallelism by drawing a graph of the parallel application. (If an X Window interface is not available, the user can input textual graph descriptions.)

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 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 arc's tail node must run before its head. 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. The first represents looping, the second conditional dependency, the third a fan-out to a variable number of identical subgraphs, and the fourth pipelining. The graph can contain loops around subgraphs that execute a variable number of times on the basis of the expression in the loop construct. Using a conditional construct, Hence can execute or bypass a section of the graph on the basis of an expression evaluated at runtime. A variable fan-out (and subsequent fan-in) construct is available while composing the graph. The fan-out's width is specified as an expression evaluated at runtime. 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 users specify the dynamic graph,

they use a configuration tool in the Hence environment to specify the configuration of machines that will compose the parallel virtual machine. The configuration tool also helps users set up a cost matrix that determines which machine can perform which task and gives priority to certain machines. Hence uses this cost matrix at runtime 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 the requested virtual machine and begins application execution. During execution, Hence automatically maps procedures to machines in the heterogeneous network on the basis of the cost matrix and the Hence graph. Trace and scheduling information saved during execution can be displayed in real time or replayed later.

The Hence environment has a trace tool that enables 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. This display illuminates icons of the active machines 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 in each procedure. The third window shows a histogram of processor utilization. Figure 5 on the next page shows a snapshot of the trace tool in action.

In addition to discovering mistakes in the graph specification, this representation helps expose more subtle aspects of the executing program, such as load balancing and network speeds. For example, the graph produced by Hence shows noticeable differences from the abstract user-specified graph. The Hence graph may expose inherent serial bottlenecks in the algorithm or a problem
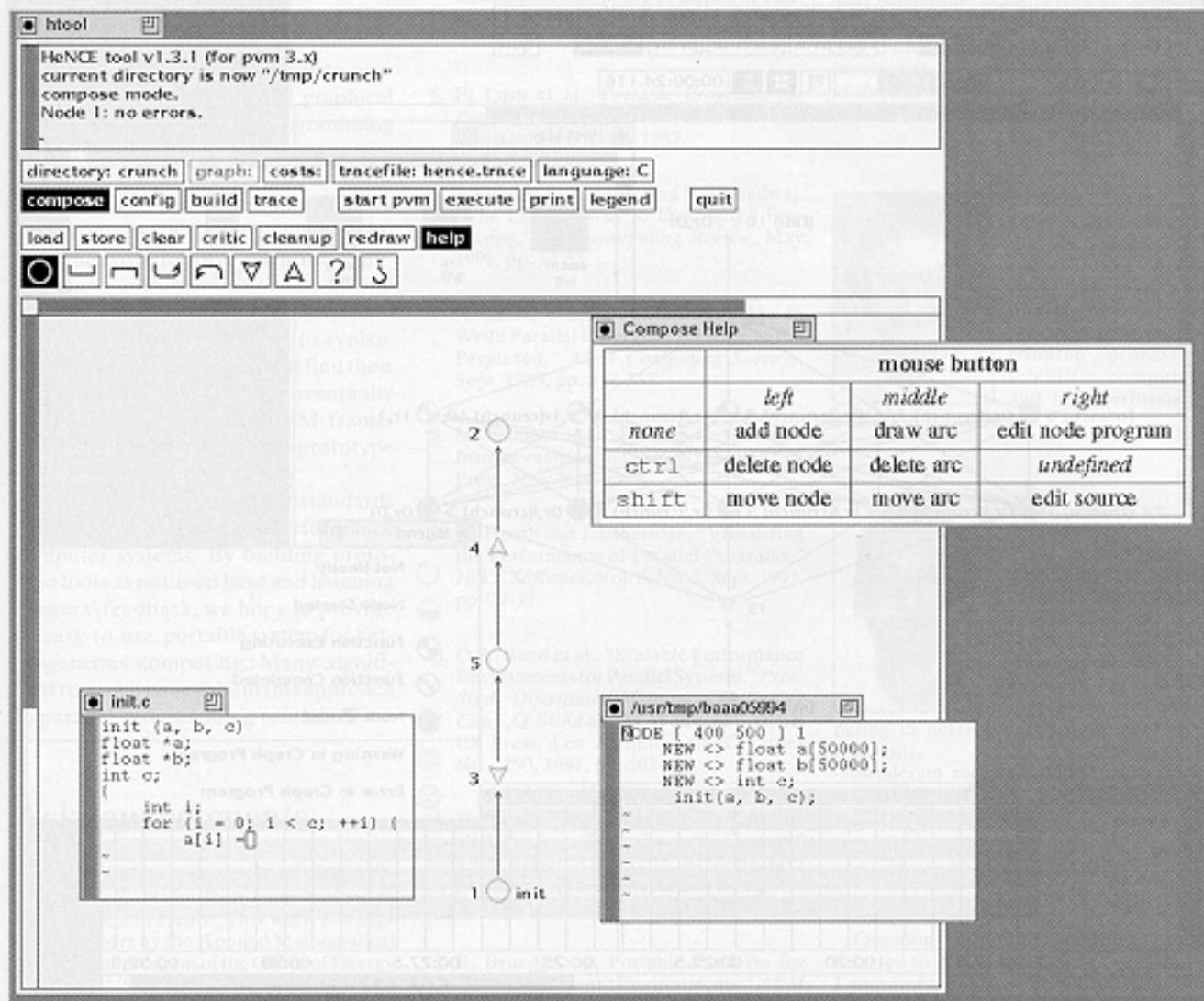

Figure 4. Composing a parallel program in the Hence environment.

with various networks used by the computation.

Our goals here are simple: to be able to schedule and trace the execution flow within an application and to understand where bottlenecks occur. In the past, users have monitored performance using a timing routine. This approach has a number of limitations in the parallel setting. We want an animation of runtime behavior that visualizes the parallel parts in execution as the application is running. We would like to know what performance issues arise during execution and what bottlenecks develop, and to see where programming errors cause a parallel program to get into trouble.

A main advantage of sequential debuggers is that they show the point of failure. In fact, many programmers resort to debuggers only when they are mystified about the point of failure, for example, 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. An advantage of the Hence trace display is that its two-dimensional display can inform the programmer of such problems. If part of a Hence program fails and

other parts continue to execute, the trace tool displays the program node failure but continues 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, and 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 program's behavior and tune it to better match the execution environment. For instance, a scientist using a network of
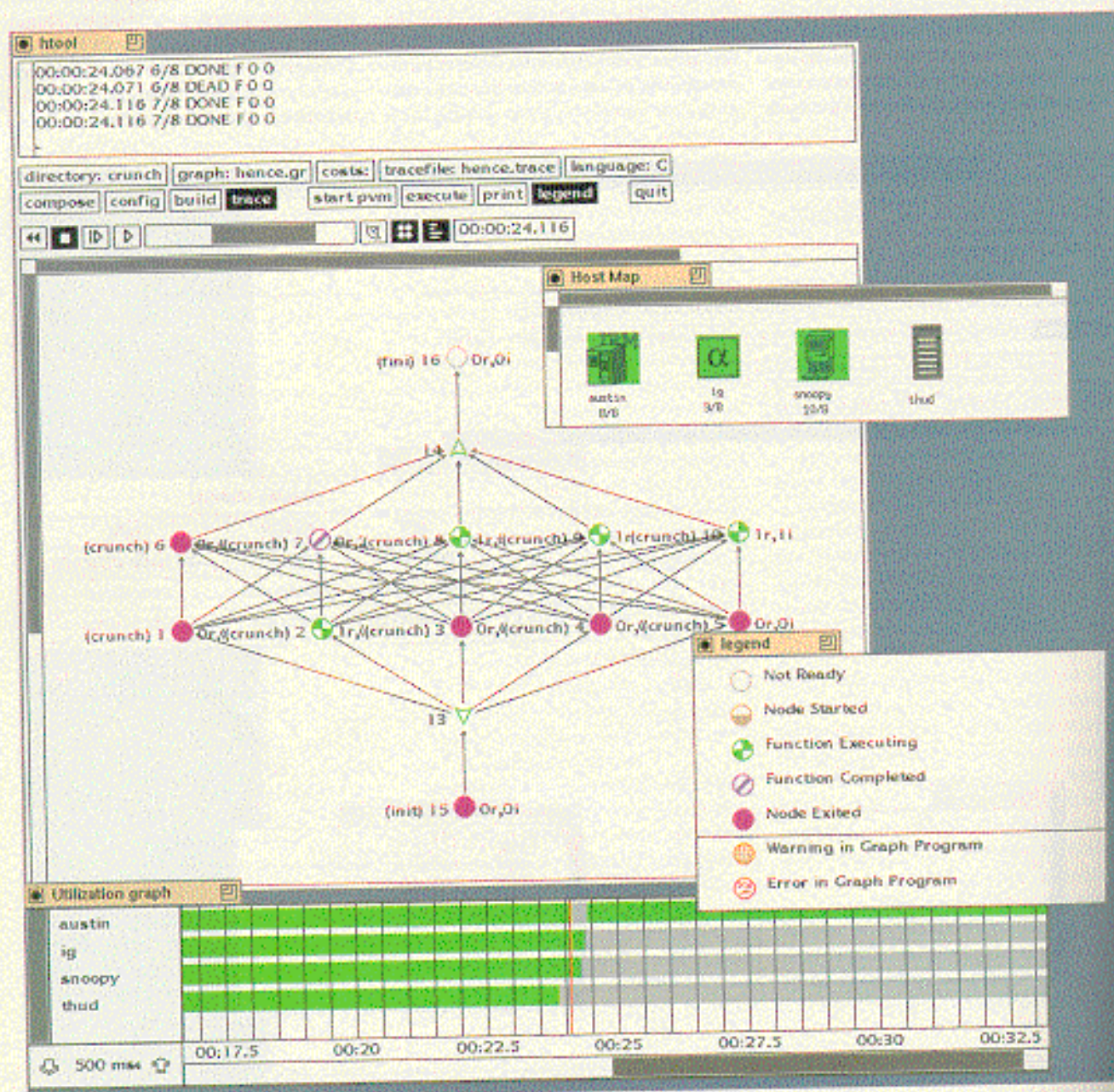

Figure 5. Tracing a parallel program in Hence.

workstations along with a Cray supercomputer may realize that during certain periods of the day it is more productive to map fewer processes to the Cray because it is heavily loaded. The Hence trace tools make this kind of information intuitively obvious. Moreover, Hence mapping is easily adjusted via the cost matrix. The user's program need not be recompiled during tuning.

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

The focus of our 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 help the user effectively use a heterogeneous computer network to solve scientific applications.

PVM, Xab, and Hence are active research projects and continue to evolve. Many of Hence's features will find their way into tools like Xab that eventually will fit directly into the PVM framework. We have already built prototype systems that are in use today.

There is a critical need for standards and tools for today's high-performance computer systems. By building prototype tools as outlined here and listening to users' feedback, we hope to provide an easy-to-use, portable system for heterogeneous computing. Many significant research issues about this approach to parallel programming remain. ∎

Acknowledgments

References

1. G.A. Geist and V.S. Sunderam, "Experiences with Network-Based Concurrent Computing on the PVM System," *Concurrency: Practice and Experience*, Vol. 4, No. 4, June 1992, pp. 293-311.

2. J.C. Browne, M. Azam, and S. Sobek, "CODE: A Unified Approach to Parallel Programming," *IEEE Software*, Vol. 6, No. 4, July 1989, pp. 10-18.

3. O. Babaoglu et al., "Paralex: An Environment for Parallel Programming in Distributed Systems," *Proc. 1992 Int'l Conf. Supercomputing*, ACM Press, New York, July 1992, pp. 178-187.

4. K.P. Birman and T.A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Trans. Computer Systems*, Vol. 5, No. 1, Feb. 1987, pp. 47-76.

5. E. Lusk et al., *Portable Programs for Parallel Processors*, Holt, Rinehart, and Winston, New York, 1987.

6. J. Flower, A. Kolawa, and S. Bharadwaj, "The Express Way to Distributed Processing," *Supercomputing Review*, May 1991, pp. 54-55.

7. N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," *ACM Computing Surveys*, Sept. 1989, pp. 323-357.

8. M. Simmons and R. Koskela, *Performance Instrumentation and Visualization*, ACM Press, New York, 1990.

9. M. Heath and J. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, Vol. 8, No. 5, Sept. 1991, pp. 29-39.

10. D.A. Reed et al., "Scalable Performance Environments for Parallel Systems," *Proc. Sixth Distributed-Memory Computing Conf.*, Q. Stout and M. Wolfe, eds., IEEE CS Press, Los Alamitos, Calif., Order No. 2290, 1991, pp. 562-569.

11. V. Herrarte and E. Lusk, "Studying Parallel Program Behavior with Upshot," Tech. Report ANL-91/15, Argonne Nat'l Laboratory, Mathematics and Computer Science Division, Aug. 1991.

12. B. Bruegge, "A Portable Platform for Distributed-Event Environments," *ACM SIGPlan Notices*, Vol. 26, No. 12, Dec. 1991, pp. 184-193.
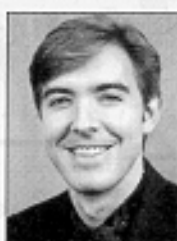
Adam Beguelin holds a joint appointment at Carnegie Mellon University's school of computer science and the Pittsburgh Supercomputing Center. His primary interests are the design and development of programming tools and environments for high-performance parallel and distributed computing.

Beguelin received his PhD in computer science from the University of Colorado.

Jack Dongarra holds a joint appointment as professor of computer science at the University of Tennessee and as distinguished scientist in the Mathematical Sciences Section at Oak Ridge National Laboratory. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced computer architectures, programming methodology, and tools for parallel computers. He was involved in the design and implementation of the software packages Eispack, Linpack, Blas, Lapack, and PVM/Hence.

Al Geist leads the computer science group in the Mathematical Sciences Section at Oak Ridge National Laboratory. His research interests are parallel and distributed processing, scientific computing, and high-performance numerical software.

Vaidy Sunderam is a professor in the Department of Mathematics and Computer Science at Emory University. His research interests include parallel and distributed processing, particularly high-performance concurrent computing in heterogeneous networked environments.

Sunderam received a PhD in computer science from the University of Kent in Canterbury, England, in 1986.