Network Based Concurrent Computing on the PVM System*

G. A. Geist

Mathematical Sciences Section
Engineering Physics and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831

`geist@msr.epm.ornl.gov`

V. S. Sunderam

Department of Math & Computer Science
Emory University
Atlanta, GA 30322

`vss@mathcs.emory.edu`

Abstract

Concurrent computing environments based on loosely coupled networks have proven effective as resources for multiprocessing. Experiences with and enhancements to **PVM** (Parallel Virtual Machine) are described in this paper. **PVM** is a software system that allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource. This report also describes an interactive graphical interface to **PVM**, and porting and performance results from production applications.

1. Introduction

Concurrent computing environments based on networks of computers can be an effective, viable, and economically attractive complement to hardware multiprocessors. A case in point is the number field sieve project of Lenstra and Manasse [1], whose most recent milestone is the factoring of the ninth Fermat number (148 digits) using over 1,000 computers worldwide. Although such large scale use of network-based concurrent computing may be rare, there exist many examples of this mode of multicomputing on a smaller scale. In all these cases, a collection of *general purpose* computer systems interconnected by *existing* networks and support services have been successfully used to achieve parallelism in applications.

* Research performed at the Mathematical Sciences Section of Oak Ridge National Laboratory under the auspices of the Faculty Research Participation Program of Oak Ridge Associated Universities, and supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

Some of these network-based concurrent computing environments are specialized, in that they are either based upon *distributed operating systems* (e.g. Locus [2], the V-kernel [3]), or they support special-purpose programming paradigms (e.g. Linda [21], the Camelot transaction processing facility [17]). While these systems are highly effective, they impose many constraints and requirements on application end-users and resource administrators that are often difficult to meet. We are concerned in this paper with the other class of distributed computing environments — those that provide general-purpose programming environments and require underlying support from the machines and their operating systems at levels that are normally considered ‘‘standard’’. As examples, programming paradigms based on the imperative model with procedure-call access to system facilities, operating system support for limited inter-process communication within a machine, and network services that provide unreliable data delivery are characteristics that such a distributed computing system would assume. Several systems that fall into this category have been described in the literature; representative examples may be found in [4,5].

In addition to utilizing available computing resources, network-based general purpose computing environments offer several other benefits. One of the most important is the potential for partitioning a computing task along lines of service functions. Typically, networked computing environments possess a variety of capabilities; the ability to execute subtasks of a computation on the processor most suited to a particular function both enhances performance and improves utilization. The Plan 9 distributed system from Bell Labs [6] is based *entirely* on this model, and initial results are very promising. But the implementation of Plan 9 appears to suffer from lack of flexibility and special requirements in terms of network characteristics and processing/storage elements. These factors imply that widespread use of Plan 9 will be possible only with a long-term and substantial commitment to the model and environment on the part of potential users, in turn implying that ubiquitous use is unlikely in the near future.

Another advantage in network-based concurrent computing is the ready availability of development and debugging tools, and the potential fault tolerance of the network(s) and the processing elements. Typically, systems that operate on loosely coupled networks permit the direct use of editors, compilers, and debuggers that are available on individual machines. These individual machines are quite stable, and substantial expertise in their use is readily available. To the user, these factors translate into reduced development and debugging time and effort, in addition to lowered contention for resources and possibly more effective implementations of the application. Yet another attractive feature of loosely coupled computing environments is the potential for user or program level fault-tolerance that can be implemented with little effort — either in the application or in the underlying operating system. Most multiprocessors do not support such a facility; hardware or software failures in one of the processing elements often lead to a complete crash.

There are, however, many aspects relating to user interface, efficiency, compatibility, and administrative issues that play a significant role in the effectiveness of network-based concurrent computing environments. In this paper, we analyze several of the design features of the **PVM** (Parallel Virtual Machine) system and report on experiences gained with its use over time. An overview of **PVM**, followed by significant aspects of the user interface and implementation strategies are presented in the following sections. Finally, representative examples of porting and performance are described.

2. An Overview of the PVM System

The **PVM** system is composed of a suite of user-interface primitives (shown in Table 2 in the Appendix) and supporting software that together enable concurrent computing on loosely coupled networks of processing elements. Several design features distinguish **PVM** from other similar systems such as Cosmic [7], Marionette [4], ISIS [22], and Dpup [5]. Among these are the combination of heterogeneity, scalability, multilanguage support, provisions for fault tolerance, the use of multiprocessors and scalar machines, an interactive graphical front end, and support for profiling, tracing, and visual analysis.

2.1. PVM Architecture

PVM may be implemented on a hardware base consisting of different machine architectures, including single CPU systems, vector machines, and multiprocessors. These computing elements may be interconnected by one or more networks, which may themselves be different (e.g. one implementation of **PVM** operates on Ethernet, the Internet, and a fiber optic network). These computing elements are accessed by applications via a standard interface that supports common concurrent processing paradigms in the form of well-defined primitives that are embedded in procedural host languages. Application programs are composed of *components* that are subtasks at a moderately large level of granularity. During execution, multiple *instances* of each component may be initiated. Figure 1 depicts a simplified architectural overview of the **PVM** system.

Application programs view the **PVM** system as a general and flexible parallel computing resource that supports shared memory, message passing, and hybrid models of computation. This resource may be accessed at three different levels: the *transparent* mode in which component instances are automatically located at the most appropriate sites, the

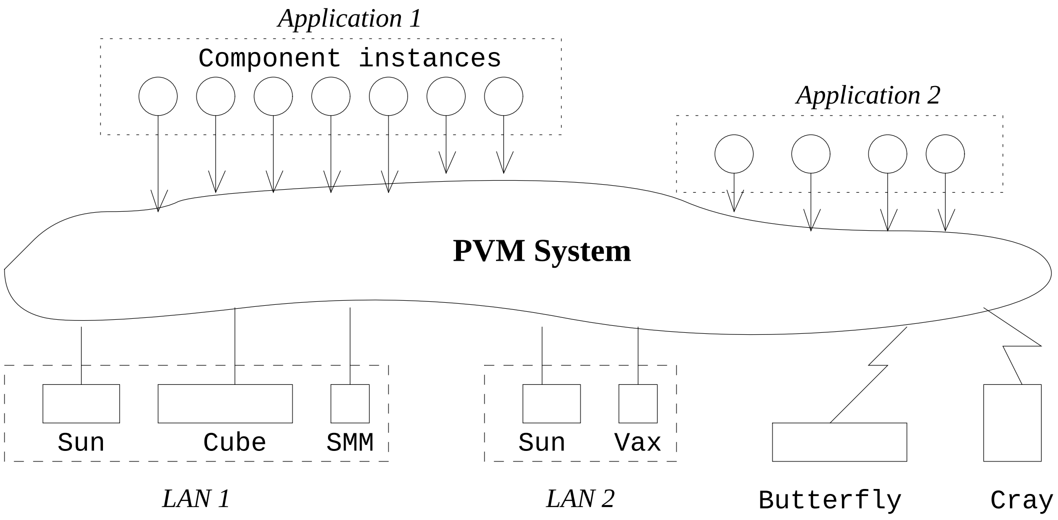

Figure 1: **PVM** Architectural Model

architecture-dependent mode in which the user may indicate specific architectures on which particular components are to execute, and the *low-level* mode in which a particular machine may be specified. Such layering permits flexibility while retaining the ability to exploit particular strengths of individual machines on the network. The **PVM** user interface is strongly typed; support for operating in a heterogeneous environment is provided in the form of special constructs that selectively perform machine-dependent data conversions where necessary. Inter-instance communication constructs include those for the exchange of data structures as well as high-level primitives such as broadcast, barrier synchronization, mutual exclusion, global extrema, and rendezvous.

PVM supports two general parallel programming models — tree computations as supported by the DIB [8] and Schedule [9] packages, and crowd computations [11]. Supporting both paradigms increases the flexibility and power of the system significantly, especially since individual subtasks within either of these models may themselves be parallel programs expressed in the other. At present, the model, individual subtasks, and their interactions are described in procedural terms; work is in progress to provide graphical specification.

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. This is the most unstructured form of crowd computation, but in practice a significant number of concurrent applications are more structured. Two typical structures are the tree and the ‘‘regular crowd’’ structure. We use the latter term to denote crowd computations in which each process is identical; frequently such applications also exhibit regular communication and synchronization patterns. Any specific control and dependency structure may be implemented under the **PVM** system by appropriate use of **PVM** constructs and host language control flow statements.

2.2. Heterogeneity Issues

The **PVM** system is heterogeneous in several respects:

- *Applications*: Heterogeneous applications are those that are composed of subtasks that differ significantly from one another. Particularly in scientific computing, there are many such applications. The components of such applications exhibit diverse characteristics including vector processing, large-grained SIMD computing, and interactive 2-D and 3-D graphics. The traditional solution to this problem is to execute each component separately on the most suitable architecture and construct manual, application-specific interfaces among them.
- *Processing Elements*: The **PVM** system is supported on various machine architectures including shared-memory multiprocessors, hypercubes, and scalar computers. In order to make the most effective use of any multiprocessors that may be available to an application, two options are provided. The first is the ability to treat multiprocessors as an atomic resource — applications may execute programs that are hard-coded for specific multiprocessors under **PVM** control; such components retain the ability to interact with other components executing elsewhere in the system. The second is the provision for dynamic incorporation of application modules in a selective manner, depending upon the architecture on which an application component executes. In the latter scheme, an application specifies several alternative modules to perform a given function, each suitable for one of the different programming models supported. At execution time, **PVM** selects the most appropriate module to utilize, depending upon the actual machine(s) on which the application will execute.
- *Networks:* Several different network architectures are supported by the **PVM** system, both for reasons of wider applicability as well as to be better able to exploit specific features of particular networks. For example, Internet protocols may be used both on the DARPA Internetwork and on Ethernets. However, specialized low level protocols on Ethernet significantly improve performance and efficiency in distributed applications. The **PVM** system presently supports the Internet protocols[11], low level Ethernet protocols [12], and the IMCS interface [13].

2.3. Other Aspects

Multiprocessing on loosely coupled networks provides facilities that are normally not available on tightly coupled multiprocessors. Debugging support, fault tolerance in the form of checkpoint-restart, uniprocessor level I/O facilities, and profiling and monitoring to identify hot-spots or load imbalances within an application are examples. On the other hand, several obstacles and difficulties are also associated with networked concurrent computing. Among these are generating and maintaining multiple object modules for different architectures, considerations of security and intrusion into personal workstations, and a number of administrative and housekeeping functions. In its present form, **PVM** supports two auxiliary components that provide some desirable features and overcome several of the obstacles. First, the XPVM interface is a graphical tool that eases many of the application tasks of specifying components, handling input and output, interacting with **PVM** during execution, managing multiple objects, and providing a debugging interface. Second, the PICL library [14] supports portable parallel programming and profiling. These components are discussed in the following sections.

The **PVM** support software (a daemon process that executes on each participating host) is replicated for each user of the system. The (small) overheads incurred are considered acceptable since this scheme eliminates many of the security and addressing issues that are encountered when common support software caters to all users. To achieve location transparency and fault tolerance, the **PVM** system uses the strategy of global knowledge among the daemon processes and identifies component instances using symbolic names and instance numbers. In the common daemon scheme, naming conflicts are possible, and further, hosts that are not used by a particular application are required to participate in all events, leading to performance degradation and delays.

The **PVM** system supports a limited form of fault tolerance at several levels. First, since individual component instances are independent processes (usually on different machines), failure of an instance does not affect others. The **PVM** system attempts to provide this level of tolerance even on multiprocessors provided that the operating system facilities permit partial degradation. In addition, individual instances that have failed may be migrated or restarted if the application so desires, subject once again to host operating system constraints.

In addition to the above, nearly all the user interface constructs provided by **PVM** contain provisions for the detection and recovery from failures, a feature rarely available as a native facility in typical tightly coupled multiprocessors. For example, to preempt some forms of deadlock, blocked message reception may be aborted either on timeouts or by placing a limit on the number of alternative messages. Barrier synchronization primitives permit the specification of a quorum of processes that are required; if it is impossible to form such a quorum, processes that invoke barrier constructs are so notified. Distributed locks may be specified as having a limited ‘‘lifetime’’; if a component instance aborts prematurely, any locks held by that process are forcibly released. While some of these facilities must be used with caution, they are nevertheless valuable — essentially, the PVM system permits applications to incorporate significant levels of fault tolerance when desired.

3. The XPVM Interface

PVM supports a wide range of facilities including the ability to configure the set of participating hosts dynamically, to debug selected component instances, to position specific processes, and to execute multiple processes that make up an application using several different control structures. These features may be used under program control, augmented by manual execution of standard utilities available on most host environments. However, in order to exploit them fully in the most effective way, a user-friendly, interactive interface is desirable and necessary. The XPVM front-end is designed to enable convenient access to the PVM facilities using a graphical interface, and is described in this section. XPVM is still evolving, but sufficient functionality is available in its present form to allow a substantial number of operations to be performed.

The XPVM interface essentially sets up an interactive ‘‘session’’ with the PVM system in a manner analogous to a login session. Sessions are on a ‘‘per-user’’ basis; indirectly, the XPVM interface permits multiple users to share simultaneously some of the support functions provided by PVM. Interaction with XPVM is accomplished via a menu-driven interface. In the remainder of this section, the functions supported by the XPVM system are described with illustrative examples extracted from an actual session.

3.1. Configuration Management

The XPVM interface consists of five major components. The first is configuration management and is responsible for managing the pool of hosts that are accessible during a session. Using this facility, PVM users may add to or delete from the pool of hosts on which a concurrent application is to execute. The configuration example shown in Figure 2 is a snapshot at the moment immdeiately preceding the addition of a transputer based machine, with hostname ‘‘cogent’’. In addition, configuration management performs authentication functions and ensures that specified hosts are indeed accessible by the user. PVM daemons are started up on each host, and information regarding the current configuration is shared among the active daemons. In addition, the daemons cooperate to assign each host an identification number for use in the execution of distributed PVM primitives such as broadcast, barrier synchronization and distributed mutual exclusion. During this phase, the PVM system also attempts to classify hosts on the basis of geographical distance, relative computing power, and load conditions. These parameters are obtained using a combination of statically defined tables and instantaneous measurements and are used during execution time to select the configuration that is likely to be most effective using simple heuristic rules.

Error diagnostics are provided in the case of authentication failure or when a specified machine or architecture type does not exist. It should be noted that additions to the host pool may be made while applications are executing; deleting a host with a live component instance causes the operation to be delayed until the instance has terminated.

3.2. Object Management

One of the most cumbersome aspects of concurrent computing in a heterogeneous network is the management of multiple object modules for each component of an application system. To assist the user in handling this issue, XPVM supports an object management interface. In its present form, this interface is somewhat limited. An example scenario showing the present facilities in the object management interface is shown in Figure 2. Work is in progress to include dictionary facilities, version control, and automatic object code generation to simplify the task of object maintenance.

3.3. Application Execution

The XPVM interface contains facilities for unstructured and regular crowd computation models. In addition, tree structured computations will be supported using the Schedule [9] system. In the regular crowd model, the XPVM interface permits the specification of an object module and the number of instances that are to be initiated; the specified number of processes are then executed automatically by the PVM system, thereby avoiding the need for a user-written driver program. In the unstructured model, it is assumed that a ‘‘host’’ program assumes responsibility for initiating the application component instances if any. The XPVM interface essentially enables the individual initiation of separate programs (each of which may subsequently spawn others) once again without the need for a control or driver program. In addition, the application execution function of XPVM permits the specification of command line arguments, as well as input and output files and redirection, either for individual component instances or for groups of processes. An example of the use of the ‘‘RUN’’ function is shown in Figure 2.

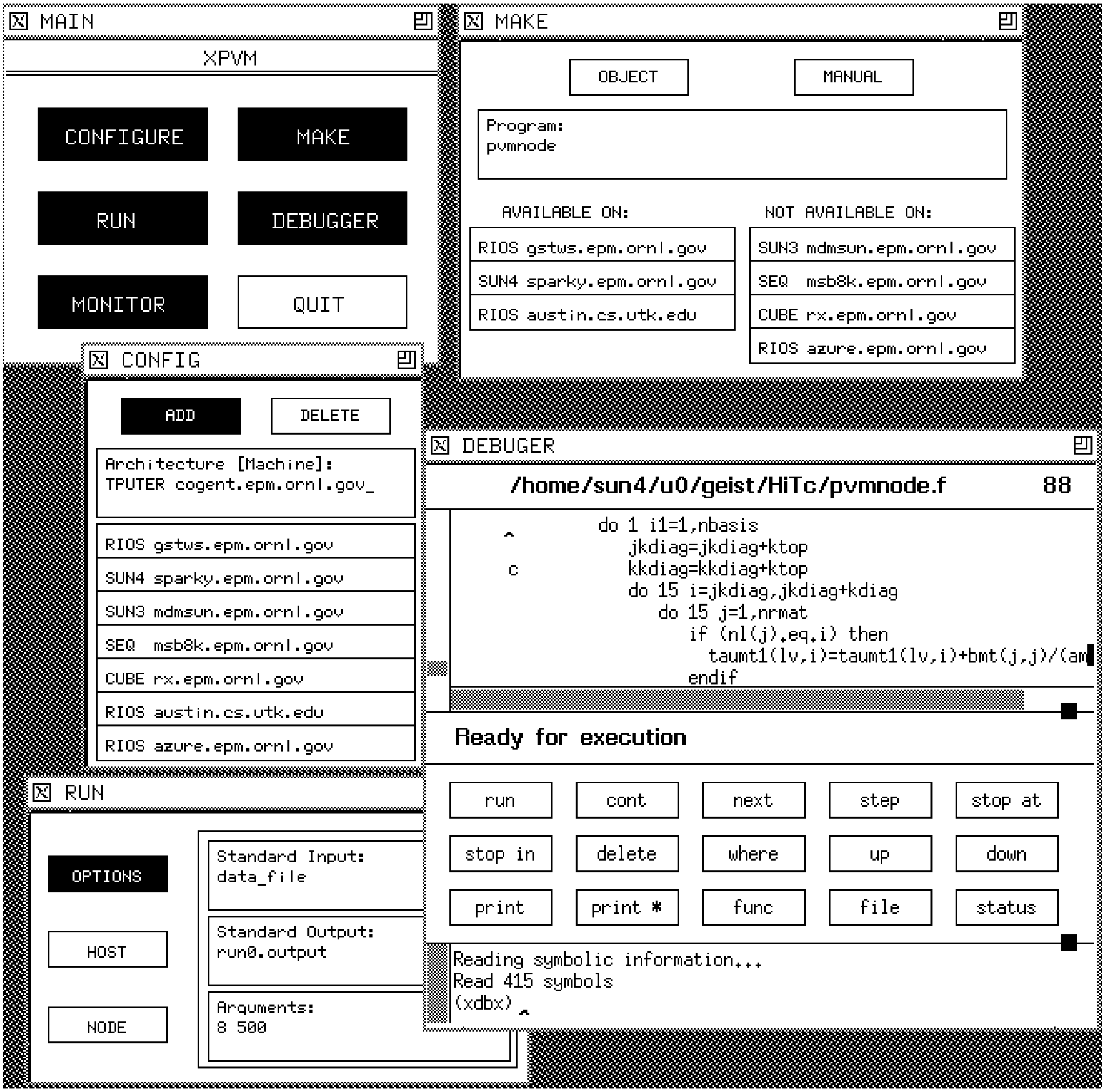

Figure 2: Sample XPVM Session

3.4. Debugging and Monitoring

One of the most attractive features of the PVM system from the user viewpoint is the ability to execute selected, individual instances of a concurrent application under control of a debugger. This facility is rarely available on tightly coupled distributed-memory multiprocessors, and its absence is a significant obstacle to rapid program development. Given this situation, **PVM** also becomes attractive as an emulator of a variety of distributed-memory multiprocessors, in addition to being useful in its own right. The XPVM interface enables interactive debugging of selected application component instances in a simple and straightforward manner. The "DEBUG" function in the XPVM front end permits the user to specify the component name and instance numbers of those processes that are to be executed under control of a debugger. When the specified instances are initiated, the **PVM** system executes them under debugger control. At present the **xdbx** debugger is used, and a separate window for each selected process is created. An alternative debugging interface that will support debugging functions for all selected processes using a single window is being investigated. Such an interface will be very valuable for actions such as simultaneous single-step execution in all selected instances. An example of the debugging interface that is available at present is shown in Figure 2.

In addition to debugging individual component instances, the XPVM ‘‘MONITOR’’ interface can monitor global events. This includes hardware status, link failures, synchronization between application instances, and communication delays. The PICL interface, described in the following section, is a major component of the monitoring function. Essentially, applications that are written in terms of this interface may optionally enable tracing, which globally logs all events including message transmission and reception, synchronization, and other distributed events. At present, these global logs may be analyzed visually using the ParaGraph tool [15], which graphically displays events, their relationships, and (indirectly) parameters such as processor utilization and load imbalances. The monitoring facility of the XPVM interface will soon be able to display event information dynamically to assist in interactive debugging.

4. Portable Programming Using PICL

PICL (Portable Instrumented Communication Library) is a collection of library routines that facilitates portable development of multiprocessor programs. A complete description of the PICL primitives may be found in [16]. The PICL libraries have been ported to the **PVM** system in order to allow applications also to be portable to a network-based multiprocessing system. The main issues in porting PICL to a heterogeneous environment are discussed in this section.

The PICL library contains a set of high-level communications routines such as broadcast, barrier synchronization, and global extrema finding. The generic PICL release implements these in terms of low-level PICL routines, thereby achieving greater portability. In the **PVM** implementation, it was found that better performance could be attained for some of these high-level functions if they were translated directly into corresponding **PVM** primitives, and therefore this approach was adopted.

One of the most valuable features of the PICL library is the ‘‘trace’’ option that permits all communication and synchronization events to be logged. Effective use of this information for performance analysis, however, is dependent on synchronized clocks on all processing elements. While clock synchronization is a problem even on machines such as commercial hypercubes, the granularity of synchronization attainable on local networks is coarser than hypercubes and continues to be an issue of concern in the **PVM** implementation. At present, a combination of the network time protocol [19] and internal **PVM** synchronization is used and is acceptable for short-running applications.

4.1. Portability in Heterogeneous Environments

Two important issues in programming for heterogeneous network environments are the issue of data representation and byte ordering. The options available are simple — the sending process either converts data to the format on the destination machine, or the sender converts data into a machine-independent (or network) format and the receiver converts from this format to the local representation. Typically, existing systems use the latter scheme (e.g. Sun XDR [18]); although conversion is performed twice, senders do not need to know the architecture type of the destination processor, nor do representations for every possible architecture have to be known at each sender.

The **PVM** system employs the following strategy — the representation that is common to a majority of the hosts in the pool is chosen dynamically as the ‘‘standard’’, and data are transmitted over the communications network in this format. Processors in the host pool that use different data representations or byte ordering perform conversions locally on both transmission and reception, thereby reducing overheads significantly and performing conversions only in a (usually) small number of exchanges. However, the present implementation performs conversions (at each end) even when two ‘‘minority’’ processors with the same format exchange data; while this could be avoided, the amount of housekeeping required in order to accomplish such an optimization is not believed to be worth the benefit.

The generic release of the PICL library is not strongly typed. Since the library was originally intended only for homogeneous environments, all communication is performed on untyped byte streams. The port of the PICL libraries to the **PVM** system therefore necessitated a few changes in both the PICL package and the **PVM** system. The PICL library was expanded to include two new routines, **pac0** and **unpac0** that perform translation of typed data to and from the ‘‘standard’’ format. This enhancement is a natural extension of the untyped **send0** and **recv0** constructs that exchange sequences of bytes. In order to provide backward compatibility, the **PVM** system also supports the untyped send and receive primitives, with the understanding that knowledgeable users might wish to execute existing PICL programs on **PVM** in a homogeneous networked environment.

Another issue in implementing the PICL library in a heterogeneous environment is the handling of various machine dependent constants, initialization procedures, and other characteristics. For example, some message passing multiprocessors require that a subset of the processing elements be allocated in a dedicated fashion to an application, while others employ the notion of processes ‘‘occupying’’ and ‘‘vacating’’ a CPU. Machine-dependent limits on the number of different message types allowed and the maximum length of each message are other attributes that must be handled. In addressing these issues, the general philosophy adopted by the **PVM** implementation is to avoid limitations wherever possible, or to use an encompassing strategy that is a superset of the limitations on existing multiprocessors. For example, the **PVM** system does not constrain message lengths, each (virtual) processing element is considered capable of simultaneously executing many component instances, and no initialization or processing element allocation is necessary. Given the general nature of the **PVM** system and the operating system infrastructure on most machines on typical networks, most of these issues are resolved in a straightforward manner.

4.2. Experiences with PICL on PVM

In order to test the PICL implementation on **PVM**, applications written using the PICL primitives were ported and tested. The porting effort required no modifications to the original programs. Performance figures for these applications under **PVM** using native constructs and PICL are compared in Table 1, which shows that the introduction of the additional PICL layer causes little or no significant overhead.

As important as this ready portability is the fact that tracing and performance visualization tools can now be used with the **PVM** system. This facility is extremely useful, and efforts are in progress to allow event logging from within native **PVM** constructs in addition to its current availability via the PICL library. To illustrate a few of the kinds of postmortem analysis possible, displays from the use of the ParaGraph tool are presented below. The application chosen is Cholesky factorization of a matrix using 8 processors and a 100x100 matrix. The experiment was run on an Intel iPSC/2 hypercube and is contrasted to a network of Sun4 workstations in the **PVM** environment. It should be noted that the granularity of this problem size is too fine to be effective in networked environments; it was deliberately chosen to highlight the value of the visualization tool in understanding

No. of processors	Problem size (Order of Matrix) 100		200		500		1000	
1	2.0	(2.0)	9.2	(9.0)	140.6	(141.5)	1046.6	(1040.2)
2	2.8	(2.8)	6.7	(6.6)	76.3	(72.3)	601.9	(603.8)
4	3.6	(3.7)	5.8	(5.7)	61.5	(60.2)	396.8	(390.5)
8	3.8	(3.5)	6.1	(6.1)	50.8	(51.2)	230.1	(228.9)
16	5.2	(5.0)	5.9	(5.7)	31.6	(30.0)	149.8	(145.1)

Table 1 : Times (in seconds) for Cholesky factorization: PICL (native PVM)

the behavior of parallel programs executing on **PVM**.

Figure 3 shows the Kiviat diagram at an advanced stage in the program's execution. This display gives a geometric depiction of individual processor utilization and overall load balance. The dark regions indicate recent utilization by shading a polygon formed by connecting individual processor utilizations, with the center representing an idle state and the circumference 100% utilization. The lighter region depicts ''high-water'' points in an analogous manner.

The diagram shown for the iPSC/2 is typical for this application on a homogeneous, dedicated distributed-memory multiprocessor. The **PVM** figure however, shows some interesting aspects. First, the marked load imbalance is evident. Second, the high-water area shows 100% utilization for *all* processors (not simultaneous) at some previous time. Both these factors are a direct consequence of greatly increased asynchrony in a networked environment, and external loads on the workstations causing their effective computing capabilities to be different.

The Gantt chart shown in Figure 4 for the **PVM** experiment also shows the asynchrony, as well as elongated busy and idle times in comparison to the iPSC/2 run. Some of this difference is also attributable to the inherent difference in processor speeds, although it is believed that the nature of the network and external factors are the primary causes. Therefore, these two diagrams in particular must be interpreted carefully in the **PVM** context, but they are nevertheless valuable for understanding program behavior and locating errors particularly when the animation is viewed.

The Feynman diagram is a display that depicts interaction between processing elements as a function of time. Processor activity is indicated by horizontal lines, while slanted lines show message transmission and reception events. This view, observed at

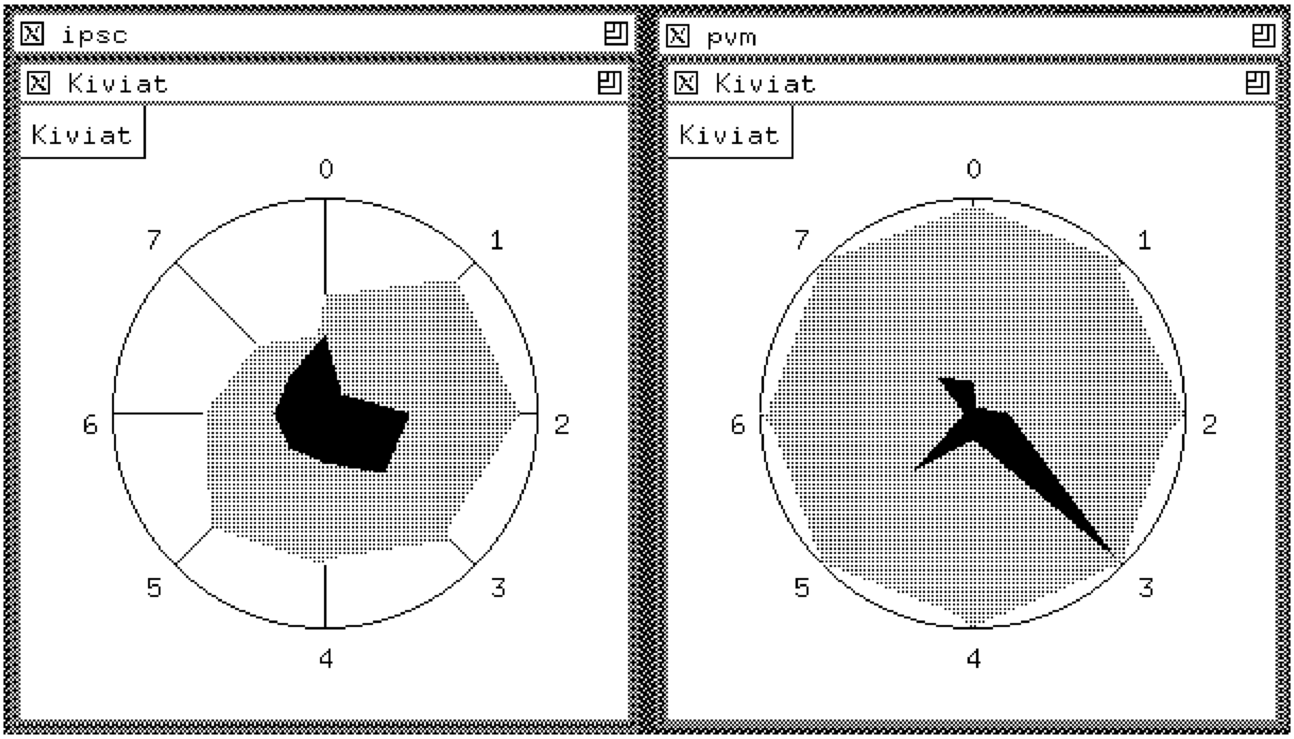

Figure 3: Kiviat Diagrams for Cholesky Factorization (100x100 matrix)

nearly identical points on the iPSC/2 and PVM, is shown in Figure 5. This example clearly displays the difference in communication speeds in the two environments, and also shows the possible variation in communication rates between the same two processing elements. Once again, this display is useful in locating bottlenecks, detecting deadlock, and as a basis for fine tuning of the application.

5. Porting Two Scientific Applications to PVM

In order to access the practicality and ease of use of PVM, two scientific applications were ported to PVM. Each application had been parallelized previously to run on hypercube multiprocessors. The size of the codes, communication patterns, and communication volumes are very different between the two applications.

Both applications are written in Fortran. This required that a Fortran-to-C interface be designed so that the PVM C functions could be called. A list of these Fortran interface routines is given in Table 3 in the Appendix.

Several problems arose during the development of this interface. The first problem was the different calling conventions of C from Fortran by different compilers. For example,

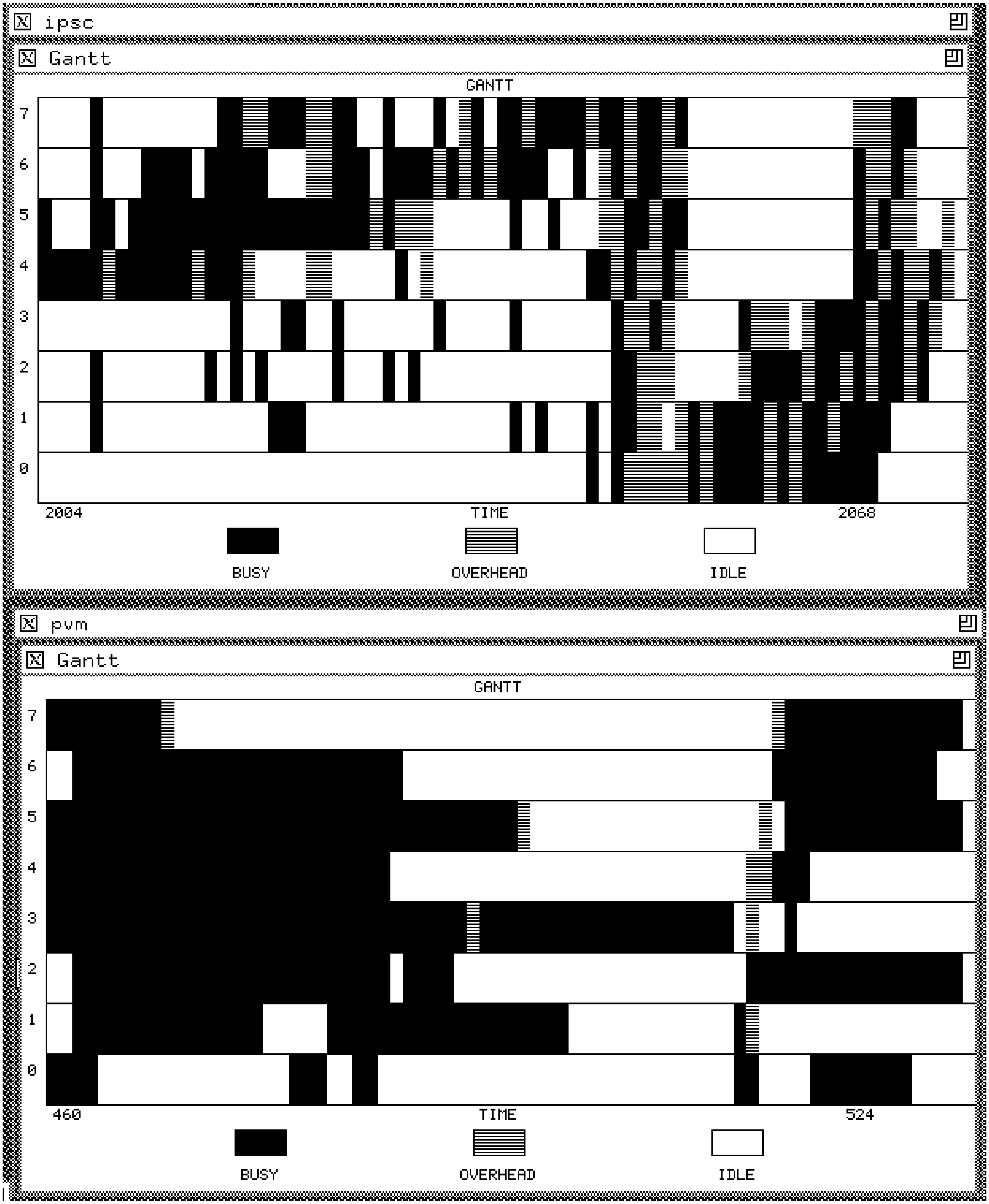

Figure 4: Gantt Charts for Cholesky Factorization (100x100 matrix)

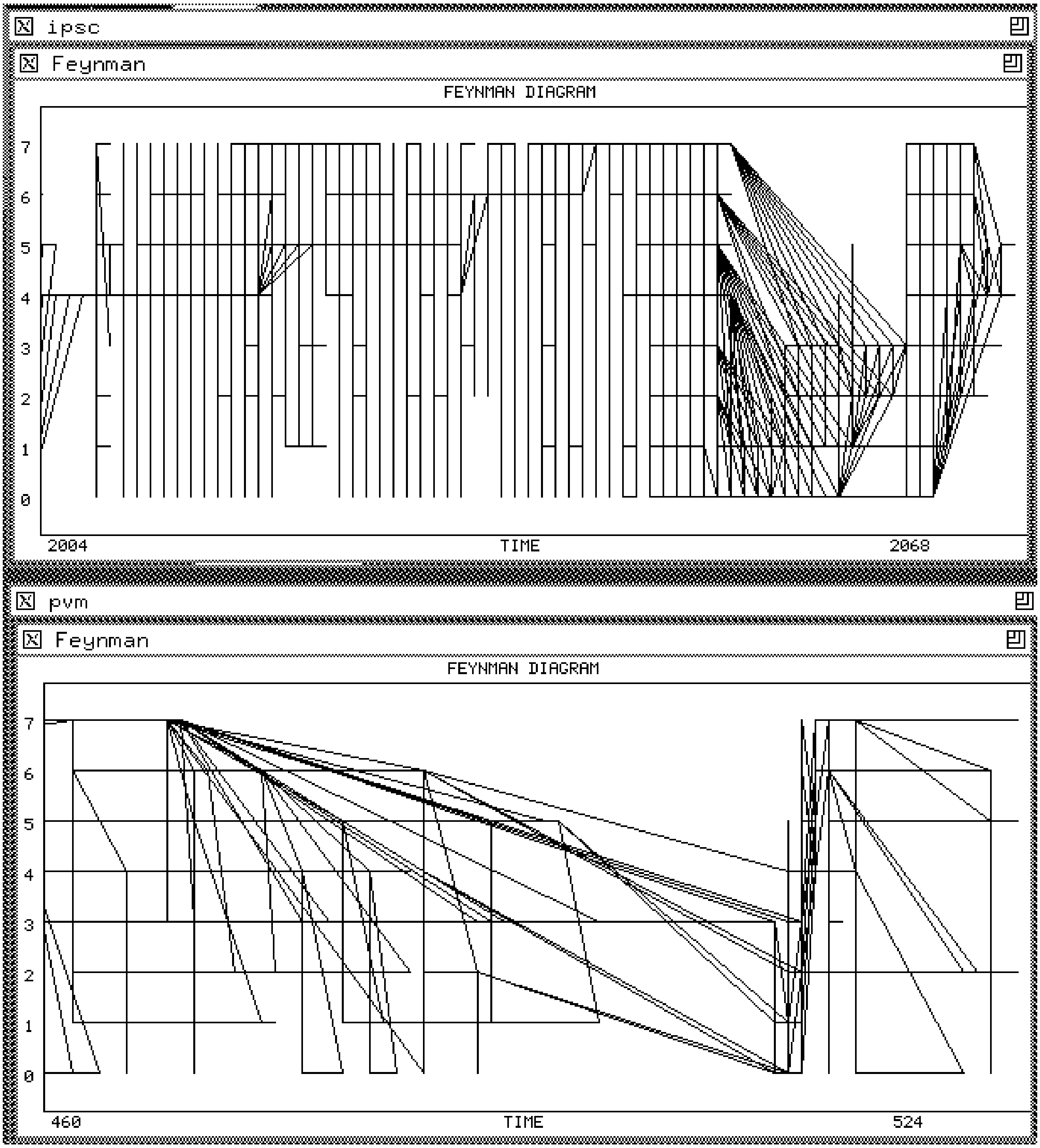

Figure 5: Feynman Diagrams for Cholesky Factorization (100x100 matrix)

some compilers prepend C routine names with underscores; others do not. This problem was resolved by having *ifdef's* for each of the different calling conventions in the interface routines. A second problem, common to Fortan-to-C interfaces, was correct passing of arguments. Fortran passes arguments by reference and C passes arguments by value.

Because of problems on some supported machines with passing values to Fortran functions, only subroutines are used in the interface. This causes the user interface to **PVM** to be slightly different when programming in Fortran rather than C. A third problem encountered was string termination. Several **PVM** routines pass strings, such as program names and signals. C terminates strings with NULLs, but this is not a requirement in Fortran so some Fortran compilers do not terminate strings. Instead, they keep track of the length of strings in an internal table. Sending a C routine a pointer to the beginning of a nonterminated string leads to nondeterministic behavior at best and a memory fault at worst. The solution to this problem requires that Fortran programmers append all the string arguments in their codes with \0. For example, *call finitiate('program\0', instancenum)*. The development of this Fortran-to-C interface was the most difficult part of porting the two scientific applications.

The first application calculates the electronic structure of metallic alloys from first principles and is based on the KKR-CPA algorithm [23]. The algorithm is parallelized using a ''Master/Slave'' paradigm in which the host process initiates tasks to perform the majority of the work. The host also coordinates the tasks to achieve good load balance. The code consists of 16000 lines of Fortran divided among 127 subroutines, but only about 20 subroutines are involved explicitly with the algorithm's parallelization.

The second application performs a molecular dynamics simulation and is used to study the interaction and vibration in molecules. The algorithm is parallelized by having multiple copies of the code solve a PDE on different spatial regions of a 3-D volume. Data are exchanged across the boundaries, and the solution is time stepped. The code consists of only 700 lines of Fortran, but nearly every subroutine is involved in some aspect of the algorithm's parallelization.

The conversion of both applications to run under **PVM** was straightforward. Changes were required in three areas. First, initiating tasks is different than in hypercubes. Tasks are initiated and then must enroll with **PVM**. Care must also be taken to ensure that tasks have enrolled before communicating with one another. If an enrolled task sends a message to a task that has not yet enrolled, the message is lost. Constructs, such as *waituntil()*, are provided in **PVM** to ensure that tasks are ready. Second, in order to facilitate the use of heterogeneous architectures, **PVM** routines are called to convert all messages to a ''standard'' format before sending and to convert them to a machine-specific format on receipt. As discussed earlier, the **PVM** routines may not actually do a conversion depending on the architectures of the sending task and receiving task. Third, sending of messages is changed to account for the fact that the user often does not know on which machine a task is running. A task (or instance) is defined by a process name and instance number. These two values are used to specify uniquely the message destination.

Having made these changes, these two applications were run on a network of Sun and IBM workstations connected by Ethernet. XPVM was used during these experiments to relieve the tedium of starting PVM on all the machines and in the case of the molecular dynamics simulation, starting each copy of the application program. Results from these experiments are given in the next section.

6. Results

The electronic structure application is computationally intensive with only a few hundred very large (10KB - 500KB) messages. Because the message traffic is small compared to the computation time, this application actually ran faster on a network of eight IBM RS/6000 workstations than on eight nodes of an Intel iPSC/860 hypercube with dedicated communication channels. The execution times for the test problem were 33 minutes and 40 minutes respectively. All of this performance gain is due to the higher execution rate of RS/6000 versus the i860 processors for this application. All 128 processors of the Intel machine have been used during computational experiments on superconductors producing execution rates in excess of 2.5 Gflops. The performance of comparable experiments on various PVM configurations of RS/6000 workstations is shown in Table 2.

Model 320		model 530	
nproc	Mflops	nproc	Mflops
serial	18.2	serial	24.4
2	31.3	2	45.9
4	63.1	4	92.2
N/A	---	7	161.9
6 (530’s) + 4 (320’s)			206.5
7 (530’s) + 4 (320’s)			226.0
1 (550) + 8 (530’s) + 4 (320’s)			261.0

Table 2 : Performance of electronic structure code
on various IBM RS/6000 configurations.

The results of the molecular dynamics application for a range of processors and problem sizes are given in Table 6. The table compares the execution times of PVM using a network of RS/6000 workstations and the iPSC/860 hypercube. Again for a small

Molecular Dynamics Simulation			
pvm procs	Problem size		
	5x5x5	8x8x8	12x12x12
1	23	146	1030
2	15	91	622
4	12	62	340
8	6	34	184
iPSC/860 procs			
1	42	202	992
2	22	102	500
4	11	52	252
8	6	27	129

Table 6: Comparing execution time (secs) for molecular dynamics application.

number of processors, **PVM** over a 1.2 MB/sec Ethernet is quite competitive with a hypercube with dedicated 2.8 MB/sec channels. Load imbalances became worse on **PVM** when eight processors were used because the workstations had different computational rates. With an even more heterogeneous mixture of machines, the load imbalances would be expected to get much worse given this application's method of parallelization. (These load imbalances are not seen in the electronic structure application because its method of parallelization employs a dynamic load balancing scheme.)

Overall the performance of these two applications show the viability of using **PVM** to achieve supercomputer performance with existing hardware. Even higher performance is expected as faster networks become available.

References

[1] A. Lenstra, M. Manasse, "The Number Field Sieve", *Proc. Symposium on the Theory of Computing*, Baltimore, May 1990.

[2] G. Popek, B. Walker, "The LOCUS Distributed System Architecture", *MIT Press*, Cambridge, 1985.

[3] D. Cheriton, "The V Distributed System", *Comm. ACM*, Vol. 31, No. 3, pp. 314-333, March 1988.

[4] M. Sullivan, D. Anderson, "Marionette: A System for Parallel Distributed Programming Using the Master/Slave Model", *Proc. 9th Intl. Conf. on Distributed*

Computing Systems, pp. 181-188, June 1989.

[5] T.J. Gardner, et.al., "DPUP: A Distributed Processing Utilities Package", *Computer Science technical report - University of Colorado*, 1986.

[6] R. Pike, et. al., "Plan 9 from Bell Labs", *Research Note*, July 1990.

[7] C. Seitz, et. al., "The C Programmers Abbreviated Guide to Multicomputer Programming", *Caltech Computer Science Report CS-TR-88-1*, January 1988.

[8] R. Finkel, U. Manber, "DIB - A Distributed Implementation of Backtracking", *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 2, pp. 235-256, April 1987.

[9] J. Dongarra, D. Sorenson, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs", in *The Characteristics of Parallel Algorithms*, MIT Press, Cambridge, 1988.

[10] G. Fox, "Parallelism Comes of Age: Supercomputer Level Parallel Computations at Caltech", *Concurrency: Practice & Experience*, Vol. 1, No. 1, pp. 63-104, September 1989.

[11] J. Postel, "User Datagram Protocol", *Internet request for Comments RFC793*, September 1981.

[12] V. Sunderam, "A Fast Transaction Oriented protocol for Distributed Applications", *Proc. Winter Usenix Conference*, pp. 79-87, February 1988.

[13] K. Rader, "IMCS Programmers Guide - Draft", *IBM Corporation*, June 1990.

[14] G. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "A Machine Independent Communications Library", *Proc. of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications.*ed. J.L. Gustafson, Golden Gate Enterprises, Los Altos, CA, pp. 565-568, 1989.

[15] M. Heath, "Visual Animation of Parallel Algorithms for Matrix Computations", *Proc. Fifth Distributed Memory Computing Conference*, ed. D. Walker and Q. Stout, IEEE Computer Society Press, pp. 1213-1222, April 1990.

[16] G. Geist, et. al., "A User's Guide to PICL: A Portable Instrumented Communication Library", *Oak Ridge National Laboratory TM-11616*, September 1990.

[17] A. Spector, et. al., ""Camelot: A Flexible Distributed Transaction Processing System", *Proc. Spring Compcon 88 - 33rd IEEE CS Intl. Conf.*, pp. 432-437, March 1988.

[18] Sun Microsystems, "XDR: External Data Representation Standard", *Internet request for Comments RFC1057*, June 1988.

[19] D. L. Mills, "Network Time Protocal (version 2) specification and implementation", *DARPA Network Working Group Report RFC-1119*, September 1990.

[20] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing", Concurrency:Practice & Experience Vol. 2 No. 4, Dec. 1990.

[21] M. Arango, D. Berndt, N. Carriero, D. Galernter, and D. Gilmore, "Adventures with Network Linda", Supercomputing Review, Vol. 3 No. 10, Oct. 1990.

[22] K. Birman and K. Marzullo, "ISIS and the META project", Sun Technology Summer 1989, pp. 90-104.

[23] G. A. Geist, B. W. Peyton, W. A. Shelton, and G. M. Stocks. "Modeling High-temperature Superconductors and Metallic Alloys on the Intel iPSC/860", *Proc. Fifth Distributed Memory Computing Conference*, ed. D. Walker and Q. Stout, IEEE Computer Society Press, pp. 504-512, April 1990.

Table 1: PVM user routines.

void barrier(char *barrier_name, int num) blocks caller until num calls with same barrier name made.
void bcast(char *component_name, int msgtype) broadcasts message to all instances of component name.
int enroll(char *component_name) enrolls process in PVM and returns instance number.
void get[type]([type] *x) extracts value of datatype [type] from received message and assigns it to x, eg. getfloat(x). [type] must be int, float, dfloat, cplx, dcplx, string, or bytes.
int initiate(char *object_file) initiates a new process and returns instance number.
int initiateM(char *object_file, char *arch [, char *machine]) initiate a process on the specified architecture [machine].
void initsend(int size) initializes send buffer of specified length.
void leave(char *component_name, int instance) process exiting PVM.
int probe(int msgtype) probe for message arrival of specified type or ’any’ if msgtype=-1. Returns message type or -1 (not arrived).
int probemulti(int num, int *msgtypes) same as probe, but permits specifying an array of num message types.
void put[type]([type] x) inserts x into send buffer in machine independent form. [type] must be int, float, dfloat, cplx, dcplx, string, or bytes.
void ready(char *event_name) sends signal with specified (abstract) name.
int rcv(int msgtype) receives a message of specified type or ’any’ if msgtype=-1 (Blocking). Returns actual message type.
int rcvmulti(int num, int *msgtypes) same as rcv, but permits specifying an array of num message types.
int rcvolim(int msgtype, int num) same as rcv, but limits the number of other messages that may arrive in the interim.
int rcvtlim(int msgtype, int seconds) same as rcv, but blocking limited to seconds.
void rcvinfo(int *bytes, int *msgtype, char *component, int *instance) returns the length, type, and sender of last received message.
void snd(char *component, int instance, int msgtype) sends message in send buffer to the specified instance of component.

void shmat_[type](char *key, [type] *ptr) attaches shared memory segment with name key to local address space at ptr for size units in typed form.
void shmdt_[type](char *key, [type] *ptr) detaches shared memory segment with name key from local address space.
void shmget(char *key, int bytes, char *flag) creates shared memory segment with name key of size bytes. Flag = (RO or RW).
int status(char *component, int instance) returns 1 if specified component is active, 0 otherwise.
void terminate(char *component, int instance) terminates a specified component.
int uinitiate(int argc, int *argv) same as initiate, but argv contains object name, arch type, machine name, and command line arguments.
int vinitiate(char *object_file, char *stdin, char *stdout, char *arglist) same as initiate, but permits I/O redirection.
int vinitiateM(char *object_file, char *machine, char *stdin, char *stdout, char *arglist) same as vinitiate, but specifies a specific machine.
void waituntil(char *event_name) suspends caller until specified signal name occurs.
void whoami(char *component, int *instance) returns component name and instance number of caller.

Table 3: Routines in Fortran-to-PVM interface.

fbarrier(barrier_name, n)
fbcast(component_name, msg_id)
fenroll(component_name, instance_number)
fgetcplx(variable)
fgetdcplx(variable)
fgetdfloat(variable)
fgetfloat(variable)
fgetint(ivariable)
fgetstring(string)
fgetstringl(string, length)
finitiate(component_name, machine, instance_number)
finitsend(length)
fleave(component_name, instance_number)
fpstatus(component_name, instance_number, istatus)
fputcplx(variable)
fputdcplx(variable)
fputdfloat(variable)
fputfloat(variable)
fputint(ivariable)
fputstring(string)
fputstringl(string, length)
fready(event_name)
frecv(msg_id)
frecv1(msg_id, other_ids)
frecvinfo(length, msg_id, component_name, instance_number)
fsend(component_name, instance_number, msg_id)
fshmat(key_name, char_buff, isize)
fshmatfloat(key_name, real_buff, isize)
fshmatint(key_name, int_buff, isize)
fshmdt(key_name, char_buff)
fshmdtfloat(key_name, real_buff)
fshmdtint(key_name, int_buff)
fshmfree(key_name)
fshmget(key_name, isize, flags)
fterminate(process_name, instance_number)
fwaituntil(event_name)