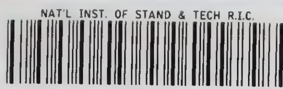
NAT'L INST. OF STAND & TECH R.I.C.
A11104 260157

NIST
PUBLICATIONS

NISTIR 5381

Distributed Supercomputing Software:

Experiences with the Parallel Virtual Machine - PVM

Richard D. Schneeman

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Systems and Software Technology Division
Computer Systems Laboratory
Gaithersburg, MD 20899

QC
100
.U56
1994
#5381

NIST

NISTIR 5381

Distributed Supercomputing Software:

Experiences with the Parallel Virtual Machine - PVM

Richard D. Schneeman

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Systems and Software Technology Division
Computer Systems Laboratory
Gaithersburg, MD 20899

March 1994

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Arati Prabhakar, Director

Abstract

The Parallel Virtual Machine (PVM) is a general purpose distributed system developed by researchers at the Oak Ridge National Laboratory and Emory University. The PVM system consist of a portable suite of software specifically designed for use by parallel and supercomputing application engineers. NIST researchers are studying PVM to assist them in defining the system service requirements needed to support parallel programming and supercomputing activities in the general purpose distributed setting. The requirements generated from this and other application domain studies will form the basis for a profile being developed at NIST that addresses these service requirements in the distributed environment. Based on the collective requirements from a variety of application areas, the NIST distributed system profile seeks to define the set of standards and specifications that all application platforms must support in order to participate in the distributed environment. This report focuses on defining the profile requirements culminating from our PVM assessment; therefore, this document will also provide reference material for those involved in evaluating distributed system software for the supercomputing domain.

Keywords. Application domain; application programming interfaces; distributed systems; graphical user interfaces; heterogeneous systems; parallel processing; portability; requirements; supercomputing.

Contents

1 Introduction **1**

1.1 Research Objectives 1

1.2 OSE Reference Model Applicability 1

2 Parallel Processing Background **2**

3 An Overview of PVM **3**

3.1 Client-Server Distributed System Model 4

3.2 Multiple Parallel Programming Models Supported 5

3.3 Graphical Tool Provides Distributed Program Support 7

3.4 Application Heterogeneity and Portability 9

3.5 Benefits of a General Purpose Environment 10

4 Capturing the Requirements **11**

4.1 Lessons Learned 11

4.2 Core Requirements Defined 15

5 Conclusion **17**

References **17**

Acknowledgements **18**

List of Figures

1	PVM System Architecture.	6
2	The HeNCE Graphical Environment.	8

1 Introduction

Distributed systems designed to support a general purpose environment must account for the wide variety of application domains they will accommodate. Accounting for this variety will inevitably shape what underlying distributed system services are required in order to support each of these domains. By characterizing the different application domains and generalizing their requirements to an Open System Environment[1] (OSE), National Institute of Standards and Technology (NIST) researchers can obtain valuable insight when defining OSE-based specifications for use in the distributed environment. By studying the systems in use by supercomputing application engineers, NIST researchers are able to discern how massively parallel applications use distributed computing techniques in the general purpose environment. Analyzing how distributed computing activities are affected by the parallel processing environment allows NIST researchers to readily differentiate between the infrastructure requirements needed to support massively parallel programming and those developed for the more traditional general purpose execution environments.

1.1 Research Objectives

The Distributed Systems Engineering (DSE) group within NIST has been researching application domain design issues that affect an OSE-based distributed system definition process. This process involves developing a Distributed Platform Profile (DPP). The DPP defines the core requirements and technology specifications needed to support a minimal OSE-based distributed computing environment. The DPP can become a citable document that procurement officials can use to facilitate purchasing decisions. The document also can provide technologists with a list of specifications that describe a technically credible engineering baseline for a generic distributed system. This research provides the partial requirements framework needed to initiate the DPP activity by NIST. Clearly, the parallel and supercomputing areas are only a small segment of the many application domains that need to be addressed in order for NIST to completely describe a full set of requirements for the DPP. This report focuses on the immediate requirements uncovered during this study.

1.2 OSE Reference Model Applicability

The NIST DPP process will use the service areas defined by the OSE Reference Model (OSE-RM) as a template in which to position the core requirement issues found. All aspects of the system under study that are relevant to a distributed system are placed into one of the following four service areas of the OSE-RM: human/computer interface, communication

[1]The Open System Environment (OSE) is an emerging framework that provides a common reference model and terminology for describing portable application software interfaces.

services, system services, or information interchange services. Any required services falling outside of these areas will be coalesced into a miscellaneous service area[2]. Final positioning of the miscellaneous requirements will be determined later. The most significant features found that place inherent requirement demands on a distributed system design will be qualified as core requirements for our OSE-based DPP. Section §4.2 summarizes the core requirements synthesized from this research.

2 Parallel Processing Background

Current research into the parallel and supercomputing areas have focused on describing computational models, parallel algorithms, and various computer architectures [4]. Less effort has been placed in the areas of software development and application support, or in the development of software construction tools. The development of software tools for both parallel and distributed computing environments is of paramount importance for successful deployment and widespread acceptance of distributed systems. Because large distributed systems are very complex, the proper tools to aid in all facets of their operation, maintenance, and management have not been generated in sufficient quantity or quality. In addition, the underlying distributed infrastructure must be flexible enough to accommodate these complexities. As information technology frameworks continue to migrate towards a distributed environment, complex software development tools and environments will be required for application profiling, debugging, version control, maintenance, and visualization capabilities. Distributed system implementations possessing these capabilities within portable frameworks will play a significant role in bolstering smoother transitions to the distributed environment.

Parallel processing systems have grown larger and become more complex than originally conceived. In addition, they require varying degrees of computer architecture and processing element interaction to achieve their desired goals. Dedicated multiprocessing architectures are not the only types of machine resources needed or available today for parallel application systems. For many applications in the supercomputing area, a typical job may require fast external input and output, high performance graphics workstation support, and traditional vector and scalar machine resources. Connecting a heterogeneous mix of machines to form a common processing pool can, on average, service most application requirements better than the more traditional high-end based supercomputer configurations that are now prevalent. In order to tap into these available resources, a software infrastructure is needed that can bridge the different architectures together while also providing a unified programming view of the computational resources. The infrastructure should provide support for the appropriate parallel programming paradigms as well as general purpose concurrent facilities. More

[2]Services likely to fall out of the currently defined OSE Reference Model definition include the services needed to support specific parallel and supercomputing activities.

importantly, the system should be capable of harnessing the power of large multicomputers and pipeline or vector architectures using their native methods, while also allowing the application to access all possible compute resources using portable techniques.

A popular system currently in widespread use that provides much of the required functionality is the Parallel Virtual Machine (PVM) software package developed jointly at Oak Ridge National Laboratory and Emory University. PVM is becoming a *de facto* development system for distributed parallel applications in the heterogeneous computing environment. PVM has similar commercially available counterparts; however, we chose to study PVM because it is available as public domain software and because it has many features and utilities similar to those available in commercial counterparts. Using the PVM system as the representative distributed system software from the parallel and supercomputing domains allows NIST researchers to readily determine the requirements that this domain places on a general purpose distributed system infrastructure. By collecting the requirements from this and other application domain studies, NIST researchers can better define a core set of requirements that will serve as input into the NIST DPP activity.

3 An Overview of PVM

From the user's perspective, the PVM system consists of a suite of user interface primitives or application program interface (API) functions. Applications use these primitives from within a familiar host language to access the underlying distributed system support software. Currently, the C and FORTRAN host languages are supported by an associated PVM-based API. The support software provides concurrent execution capabilities across networks of loosely coupled computers.

Single CPU systems as well as large multiprocessing machines are viewed as general purpose (virtual) parallel processing nodes or *computing elements.* Computing elements are physical nodes representing hardware execution facilities. *Processing elements*, are instances of intra-application modules spread out over a network of computing elements. Processing elements are further decomposed into *components*, which are subtasks or procedures representing a high-level of parallel application granularity. Components within processing elements are synonymous with procedures found in traditional programming languages.

The PVM heterogeneous environment includes design features to support heterogeneity, scalability, multi-language support, and fault tolerance of application software. Communication and synchronization constructs for exchanging data structures between application instances are provided by way of a specific portable support library. The PVM system includes a library specifically tuned for portable software development across a variety of multiprocessor architectures.

The Portable Instrumented Communication Library (PICL), works in conjunction with

the PVM system and provides portable multiprocessing facilities in a networked heterogeneous environment. Primitives from this library provide process spawning, message passing, and shared memory capabilities that resemble the methods used in traditional multiprocessing based programming environments. With the incorporation of these methods into PVM, a high degree of portability and backward compatibility with a familiar programming paradigm is maintained. The PICL parallel programming library also includes support for multiple parallel computational models and high-level communication routines that provide broadcast, barrier synchronization, global extrema finding, and tracing capabilities. The tracing functions provide facilities to log all communication and synchronization events occurring in the system. Monitoring application performance gradients is also possible, provided each processing element has been internally synchronized via a timing source. The network time protocol (NTP) as well as internal PVM functions are currently used to provide the timing source capabilities.

Computing elements are accessed through PVM by using three methods or modes of interaction. Depending on the mode and type of application, the programmer has the ability to choose the degree of transparency desired. *Transparency* refers to the methods used by the software infrastructure to shield the programmer from the distribution details in the system [7]. Traditionally, transparency issues have involved the following areas: access, location, migration and concurrency of application entities. PVM views transparency methods differently, offering a less constraining definition of transparency in order to facilitate specific programmer interaction and intent. The transparency methods available to the programmer are divided into three modes: a *transparent* mode, in which the PVM system delegates placement of each processing element to any available node it deems appropriate; an *architecture-dependent* mode, which allows the programmer to explicitly place elements on architecture specific platforms; and lastly, a *low-level* mode, which provides an opaque view of the system by allowing direct placement of an element to any particular type of hardware. Locating processing elements on nodes using the low-level method of placement requires considerable target environment knowledge in order to distribute the application. It is this detailed information about the target architecture that enables the programmer to exploit specific classes of machines available on the network.

3.1 Client-Server Distributed System Model

Many distributed system modeling techniques have been used to construct a variety of software infrastructures. Distributed operating systems, transaction processing systems, and highly specialized environments for multicomputers use some form of underlying distributed modeling techniques. Hybrid approaches exist as well, leveraging specific distributed architectural design aspects and tradeoffs from a multitude of research areas. Modeling techniques range from those based on the client-server style of application interaction to service requests

based on peer-to-peer messaging techniques familiar to object-oriented systems. The PVM system addresses the former model and is described as being a general purpose client-server programming environment requiring only standard operating system support. The assumptions made by the PVM architecture provide the service framework needed to support the client-server style of application interaction and the subsequent model for the inherent distributed system. The standard operating system services PVM assumes include: (1) using a programming paradigm based on an imperative, procedure-call method of requesting services, (2) an operating system that inherently supports interprocess communication, (3) an unreliable data delivery mechanism when using a network source. These assumptions provide NIST researchers with important information as they begin the core requirements definition process. As the essential requirements from the PVM study are encountered, they will be catalogued and referenced as OSE related distributed system issues. For example, the standard operating system services PVM requires can be positioned into the OSE environment definition. The first two requirements belong in the internal system services area, which includes support for procedurally defined programming languages and local interprocess communication primitives, respectively. The third requirement clearly requires a communication service framework in which a network medium takes part. Figure 1 illustrates the client-server design of the PVM system as several PVM daemons[3] execute across a trio of heterogeneous machine architectures, identified in Figure 1 by the grey borders[5]. The PVM daemons are the principal system servers in the PVM client-server architecture and are responsible for all coordination, dispatch, and reception of messages to and from host nodes in the system. PVM daemons are also responsible for initiation of all processing element execution activities on each computing element. Network connectivity between PVM servers and clients involves the use of several defacto protocol suites. These include the Transmission Control Protocol (TCP), the Internet Protocol (IP), and the User Datagram Protocol (UDP) suite for all its socket-based routing and messaging interactions.

3.2 Multiple Parallel Programming Models Supported

Several computational models exist for programming parallel systems. Two methods used in the PVM system to represent methods of concurrency within applications are the tree and the crowd structures [6]. *Crowd* computations are used when each process is identical. These types of computations exhibit similar communication and synchronization patterns. *Tree* computations are represented by applications in which sub-processes communicate and synchronize with each other in order to coordinate and reach a desired goal. Both types of dependency structures can be implemented using a combination of appropriate PVM

[3]Daemon refers to UNIX-based server components in a client-server architecture as defined in the BSD UNIX networking parlance.

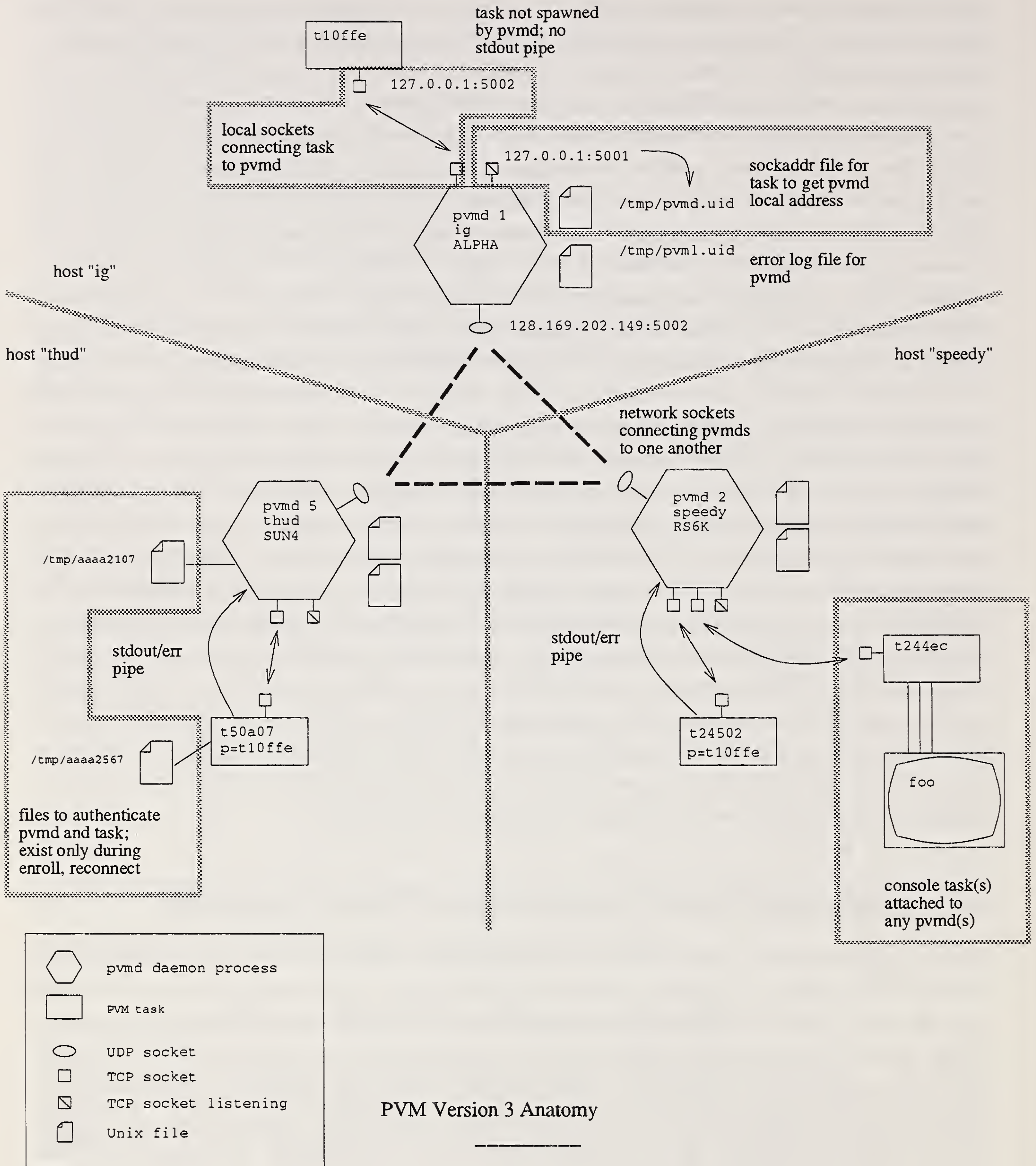

Figure 1: PVM System Architecture.

and host language statements. PVM can be programmed using either paradigm, while also supporting the capability to intermix the models within each application component. Such flexibility greatly enhances the programmers environment. In the PVM system, all models, subtasks, and their interactions are described in procedural terms. Applications at some point may require different control flow, communication, synchronization, and dependency structures. Using the constructs provided by PVM in association with the programmers host language, many different logic and control flow configurations can be used in order to express the problem in terms of PVM. Multiple host languages can be used to implement different parts of the same logical application. For example, using a component description file[4], a node has the ability to execute an application procedure implemented in FORTRAN while spawning yet another logical procedure implemented in the C language.

3.3 Graphical Tool Provides Distributed Program Support

A graphical user interface (GUI) component called the Heterogeneous Networked Computing Environment (HeNCE), provides capabilities that support application software generation, profiling, tracing, visual analysis and management of the PVM distributed system. The HeNCE tool internally uses the PVM system services of the infrastructure to provide the ability to create, distribute, execute, visualize, and maintain application programs on behalf of the user.

The HeNCE graphical tool shown in Figure 2 provides an X Window-based graphical alternative to the procedurally described component interactions within the PVM system. Shown in *compose* mode, Figure 2 illustrates how the HeNCE tool allows the programmer to sketch the nodes representing the intra-application component procedures. Data flow pathways among the computational components are provided by connected arcs drawn between each processing element. This results in a visual processing graph that when executed will animate the progress of the application's computational state.

The HeNCE tool uses the graphical depiction of the algorithm to generate the skeletal distributed software stub routines required to distribute the selected application across the network of cooperating nodes. The programmer then fills in the resultant stub shells with the application specific algorithms. The idea of using stub code comes from the use of the remote procedure call paradigm [1]. *Stubs* refers to the software produced by wrapping the local procedure calls and their actual parameters with network compatible code. The subsequent software has been marshalled[5] into a form that is architecture neutral and suitable

[4]A component description file (CDF) describes the application name, location, object file name, and hardware architecture of each participating component. The CDF is read into PVM after the startup process has been initiated, or manually using the X Window-based HeNCE tool environment.

[5]Marshalling refers to the process of translating the actual parameters in a procedure or function call to the network specific data representation.

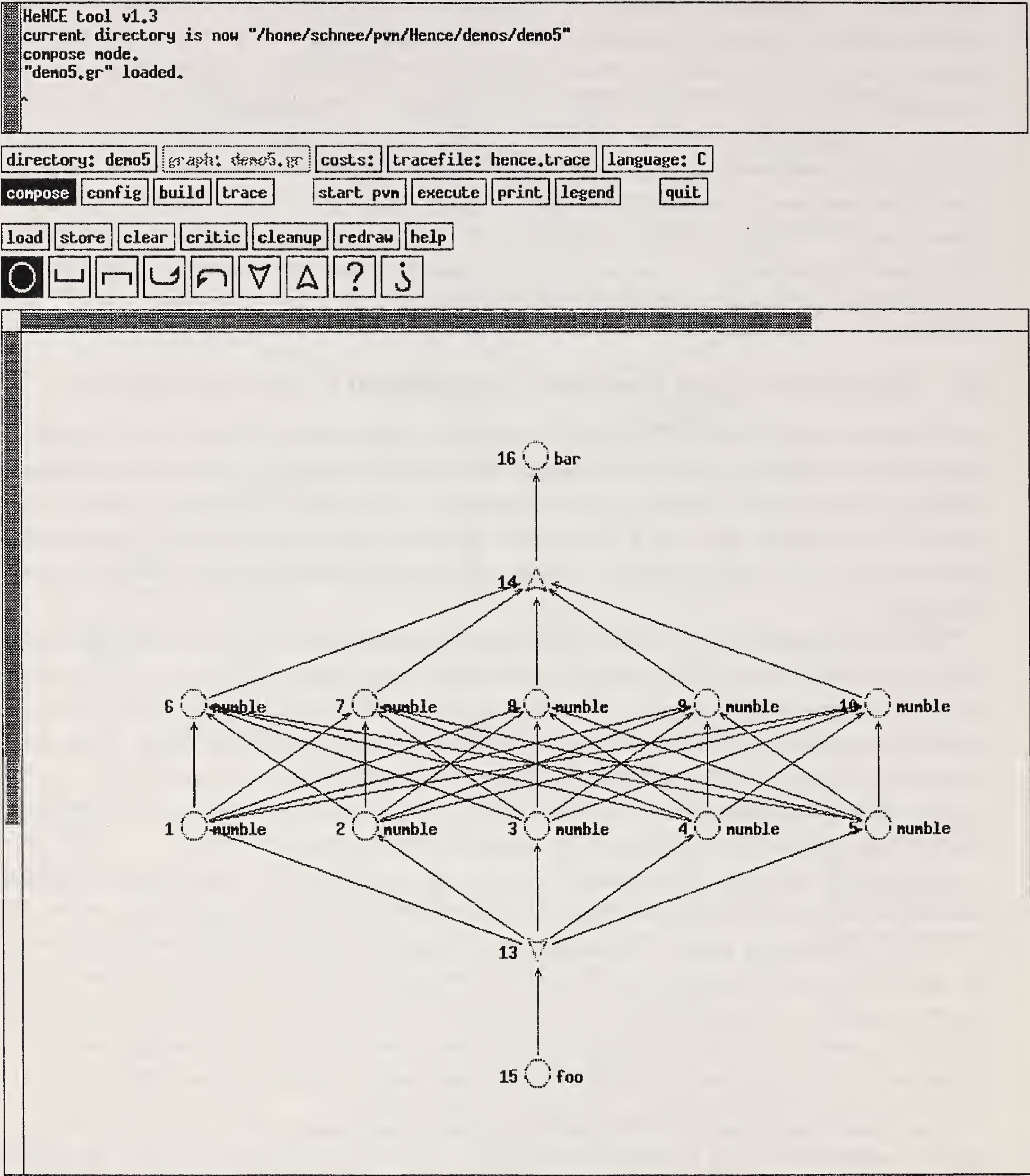

Figure 2: The HeNCE Graphical Environment.

for transmission over a network. On the remote side, the reverse process takes place and the network-based function call and arguments are un-marshalled into the remote machines particular call structure and executed. This process allows the familiar local procedure call paradigm to naturally migrate into the network environment without modifying the sequential application development paradigm that programmers have become accustomed too. The stubs then become templates for the programmer to insert the specific algorithmic requirements of the application software. Therefore, the network-based software needed to implement the distributed portions of the currently defined application are automatically generated for the programmer. This alleviates the need for time-consuming network-based code development efforts and allows the programmer to focus completely on the specific parallel activities of the application software. In addition to generating the distributed infrastructure support code, the HeNCE tool combines facilities for software development, deployment[6], maintenance and versioning of the software code base. These capabilities combined with a graphical environment render this an indispensable productivity tool for distributed software development. An interesting facility provided by HeNCE is its ability to visualize the computational state of the executing algorithm across the PVM topology. Each node's computational state is graphically depicted and displayed from within the HeNCE environment. In addition to these visualization techniques, the tool also maintains logs of all activities and events occurring within the system. The log files can later be used to playback, analyze or provide demonstrative capabilities of the executing application. More importantly, the ability to re-create the application's execution path allows the programmer to debug and profile the application in order to pinpoint semantic inconsistencies or to locate possible bottlenecks during the application's execution cycle.

3.4 Application Heterogeneity and Portability

Differences in application platforms require intervening techniques in order to provide a portable bridge across the services they provide. *Heterogeneity* refers to the methods programmers use to develop software using these techniques in order to achieve application portability across dissimilar platforms. The references to heterogeneity would normally be considered methods for achieving application portability. However, in the PVM system, heterogeneity is defined in a variety of ways. For example, *application* heterogeneity refers to a way of describing applications comprised of logically related subtasks; however, the processing portions of these subtasks are in fact executing completely unrelated activities. This occurs when applications spend large amounts of CPU time doing 2D and 3D graphics, vector processing, and coarse-grained Single Instruction Multiple Data (SIMD) style parallel

[6]The term deployment refers to the ability of the HeNCE tool to generate and deploy object-code based representations of specific application instances throughout the currently defined topology of participating architectures.

processing from within the same application. *Processing* heterogeneity among processing elements refers to an application's ability to execute hard-coded program instructions destined for a specific multiprocessing architecture while still retaining the capability to interact with higher-level PVM modules in the system. Another form of processing heterogeneity allows the application to clone several versions of a module suitable for a variety of architectures. At runtime, PVM determines what machine to execute the module on based on a free node and machine availability algorithm. *Network* heterogeneity refers to the ability of PVM to operate over a variety of networking topologies. Section §4.1 refers to this topic in more detail as this is an area of considerable interest and debate. In short, the network heterogeneity issue needs to be addressed further by the PVM team to truly be considered heterogeneous. The API concept is a portable one; however, the current PVM source code definition does not allow for the use of an Open Systems Interconnection (OSI) based transport, as complete network heterogeneity would imply.

Issues involving data representation and byte ordering have plagued heterogeneous distributed system designers for some time. In the PVM system, the strategy used to determine the proper data representation uses a majority function. The predominant data representation that is used in the current topology of computing resources is designated as the *majority* way to represent data across the network. This means that minority processors must first locally convert their data to the corresponding majority representation prior to any network-based compute resource interaction. An optimization technique that would allow minority hosts to eliminate this conversion was not implemented due to an increase in housekeeping overhead, which would not offset the net gain in performance as a result of the decrease in marshalling activities. Other issues dealing with heterogeneity include handling machine dependent constants and initialization procedures for the various multiprocessor architectures. The PVM infrastructure considers these issues in a non-restraining *status quo* fashion, thereby, minimizing portability risks.

3.5 Benefits of a General Purpose Environment

Several benefits arise from using the PVM general purpose distributed computing environment for the parallel and supercomputing domains. The PVM software infrastructure provides the ability to partition application subtasks into specific service requirement areas in order to facilitate execution of processing elements on machines particularly well suited for that task. In addition, large processor pools can be formed from high-end compute workstations while using available high performance graphics workstations for the applications visualization and GUI requirements. High speed input and output connections to fast striped or log structured filesystems can be used for large contiguous volumes of data. General purpose workstation environments typically contain an extensive workbench of stable and familiar software development tools. PVM allows that familiar set of devel-

opment tools to be incorporated as part of the HeNCE tool environment, providing for the rapid prototyping, turnaround, and deployment of distributed software. User or program-level fault tolerant mechanisms are also more conveniently supported in this environment. Multiprocessing systems will typically halt or crash when programming errors are introduced. However, a networked distributed set of compute nodes incorporating its own software-based fault tolerant strategies has the ability to restart, migrate, or simply terminate the entire application sequence if semantic errors are introduced. In the case where a computation is either long running, input/output bound, or requires checkpointing, added flexibility is introduced by allowing the job to continue using an alternative processing strategy based on the participating PVM resource topology available at the time.

4 Capturing the Requirements

The process of collecting the OSE-based distributed system requirements for the NIST DPP has been an ongoing and cumulative one, spanning the entire research, installation, system build, system management, application development, and execution phases of the PVM study. Subsection §4.1 highlights the issues and "lessons learned" during our multi-phase study of PVM system. The core requirements presented in Subsection §4.2 have been synthesized from the "lessons learned" Subsection §4.1. The requirements analysis process provides the core requirements necessary to support parallel and supercomputing in the general purpose distributed environment. In addition, peripheral issues relating to the basic support of applications in the distributed environment have been raised as a result of studying PVM.

4.1 Lessons Learned

After researching, developing applications, and working with both the PVM system and the HeNCE GUI-based tool, we have compiled a number of "lessons learned" that we believe will provide invaluable feedback into the NIST profiling activities. The lessons learned stem from direct observations with using the PVM system, accessing the network-based software from within applications, and by working with the window-based HeNCE tool. Several issues remain open-ended and do not have a one-to-one mapping with the requirements generated. The issues listed include advantages, disadvantages, positive aspects, and limitations of the PVM system. They are listed here for completeness and to assist NIST researchers in drafting the DPP and other future distributed system requirements. The important lessons learned derived from our PVM study include:

1. C and FORTRAN are used as the preferred application host languages. Support for the C and FORTRAN API indicates that a majority of the code base for the parallel

and supercomputing domains has been written in these languages. Not unlike many other domains, language portability is a major concern.

2. TCP/IP and UDP are used as the preferential networking transport medium. The BSD UNIX socket abstraction is used as the underlying network model for PVM daemon communication.

3. Standard assumptions include the use of operating system calls and interprocess communication primitives that are inherently available across a multitude of platforms. This points out the ubiquity of the TCP/IP protocol suite, POSIX style operating sytem functionality, including local IPC constructs.

4. The network heterogeneity issue assumes that the underlying PVM system is portable to a variety of network topologies. This is not the case as the PVM system software uses the BSD UNIX socket abstraction to facilitate the network-based interprocess communication between processing elements. Therefore, it is the case that the PVM API would be portable over a variety of network implementations; however, the actual PVM daemons as well as the network-based client support software would all need to be re-engineered for each particular type of network transport API used. For example, re-engineering the PVM system to the OSI stack at this time would entail using an OSI compatible transport API based on the OSI ACSE/Presentation Application Program Interfaces from the IEEE P1238 Working Group of POSIX. Alternatively, the system software could use the remote procedure call standard that is currently in the standards process at ISO. Both paradigms for re-engineering would clearly require modification of the PVM infrastructure routines that provide the networking support for PVM. In addition, the client-server daemon structure of the system software code base would also need to be re-engineered for the new API. Due to the network heterogeneity issues, the difficulty of scalability to alternative networking topologies such as OSI needs to be considered.

5. There are considerable tradeoffs between the levels of application granularity and system performance. When the level of granularity is too small (i.e., the component size of intra-application procedures is small) then system degradation occurs because of the increased overhead resulting from the emulation of multiprocessing style shared memory and message passing primitives. Marshalling parameters combined with network latency delays contribute to the consumption of inordinate amounts of resources, eliminating the benefits of partitioning the application for general purpose parallel activity in the first place. Therefore, a proper component size is needed to allow the currently executing graph of processing elements to adjust accordingly. A large

component size decreases marshalling overhead yet simply reduces the computation to that of the original application executing in a uniprocessor-based arrangement.

6. Another performance issue concerns the participation of hosts in the messaging of events and occurrences. Hosts that are not responsible for specific mediation in certain participating application topologies are still required to respond to all events; thereby, degrading overall system performance. In this case, some low-level filtering of events using small performance gradients are necessary in order to minimize the total network traffic and daemon processing overhead.

7. Bare multiprocessing machines do not typically support debugging, fault tolerance, input and output facilities, profiling, and monitoring of applications programs or intra-application components. Therefore, by using a software infrastructure such as PVM, all of these facilities can be supported by higher level software-based constructs.

8. PVM supports facilities for generating and distributing multiple copies of objects files for a variety of architectures. Without such facilities in place these tasks are tedious and in general compound the problems associated with distributed system configuration management and maintenance issues. PVM addresses these management issues by providing a software-based graphical distributed management tool.

9. All aspects of security, workstation intrusions, and data integrity have not been addressed adequately. A PVM daemon executing on each node alleviates some of the addressing and security issues; however, the underlying security risks of the UNIX environment are ever present. A security software layer would need to be placed around the PVM architecture to encase the system software infrastructure and protect it from malicious calls from the application and other vulnerabilities of the UNIX environment.

10. System administration, management, and housekeeping activities are extremely complex in the distributed environment. Tools and utilities need to be supported in order for administrators to be able to adequately manage and maintain the distributed system, its tools, the object code, and the users' software projects. Some connections with existing software development environments have been made, with a large gap in the management and deployment of software modules and user preferences profiles.

11. The HeNCE graphical tool minimizes many of the complexities involved in distributed system management issues, such as starting individual PVM daemons on all participating machines, object code placement, and software development tool integration. However, this still does not alleviate the important security and addressing concerns. Providing these types of tools is of paramount importance to successfully deploy future distributed systems on a large scale. The lack of sufficient tools becomes a critical one,

leading to the explanation of why de facto distributed systems have not been deployed on a large scale; the tools and software development environments for these systems are just not available, or are extremely proprietary.

12. The Portable Instrumented Communication Library (PICL) provides a portable library for allowing access to parallel programming systems. Ported to PVM, this library provides a portable interface to multiprocessing machines in the networked environment. A PICL type of library should be included as part of all general purpose distributed computing library frameworks. This will allow an even greater portability of application code to multiprocessor machine architectures.

13. Naming conflicts and possible global knowledge among daemon processes used to identify components with symbolic names and instance numbers can occur, rendering the system in an unknown or dangerous state of execution. This relates to the issue of scalability to other networks, larger cells, and global environments.

14. Detection and recovery from failures is provided by PVM, but not so with native multiprocessing systems. Software-based fault tolerance strategies not provided by native multiprocessing systems can be implemented using PVM constructs. Provisions for software fault tolerance in PVM include:

    - failure of an application instance will not affect any other executing node,
    - fault tolerant procedures are available to multiprocessing machines that participate and use the PVM infrastructure, and
    - restarts and migration of failed instances are possible by the application software.

15. Deadlock pre-emption techniques can be applied by: (1) aborting blocked messages using timers or message limits, (2) using barrier synchronization techniques, or (3) distributed locks can be used to prevent or handle deadlock situations. All these provide an appropriate level of fault tolerance to the application.

16. PVM provides support for a variety of computational models and programming paradigms. This provides the programmer with enough flexibility to express virtually any problem using a mix of appropriate parallel and computational models inherently provided by PVM's associated support libraries.

17. It is important to have a standard windowing environment that will support graphical tools that are capable of:

    - Creating applications without indepth programmer knowledge of the required distributed infrastructure support code.

- Provisions for distributed system stub code generation.
- Automatic placement of application instances on a pre-defined node topology.
- Automatic initiation of the PVM daemons and other requisite system software.
- Automatic object-code placement on the designated node topology.

4.2 Core Requirements Defined

The core requirements that have been synthesized from the "lessons learned" Section §4.1 are discussed here. The requirements uncovered from the parallel and supercomputing domains have been brought forth from this study and will form one of several threads that provide input into the NIST DPP profiling activity. Using the OSE-RM defined service areas as placeholders, we begin to position the core requirements into each respective service area that most accurately reflects the distributed system attributes found during our study. Four interface service areas from the OSE-RM are used to "collectively" represent the minimal set of requirements for distributed services. These requirements have been cultivated from the "lessons learned" Subsection §4.1 include:

1. **Human/Computer Interface Services**

   Human-computer interface (HCI) services define the methods by which people may interact with the application and the environment. Some of these requirements include:

   (a) A method that allows humans to interact with the application platform using command language and/or graphical user interface capabilities.

   (b) A common representation for specific graphical user interface appearance and behavior.

   (c) A common representation for specific command line user interface appearance and behavior.

   (d) Common methods and protocols for applications that utilize graphical user interfaces to interact between application platforms.

   (e) A full-featured programming language binding to the graphics portions of the application programming interface of the graphical user interface routines.

   (f) A full-featured programming language binding to graphical user interface services made available to the application.

2. **Communication services**

   Communication services provide the capabilities and mechanisms to support distributed applications requiring data access and applications interoperability in heterogeneous, networked environments. Some of these requirements include:

(a) A protocol stack that will allow communication interoperability across distributed application platforms.
(b) An application programming interface that will allow portable access to the networking services provided by the underlying communication infrastructure.
(c) Protocol independent application program access to the network.
(d) Facilities that provide some form of global naming services to cross-platform application software entities.

3. **System Services**

System services are the core services needed to operate and administer the application platform and provide an interface between application software and the platform. This interface may be divided into two types of specifications; i.e., Language Service and System Services API specifications. That is, the programming language support on the platform is represented here and the system service specific API calls are all together in this area. Some of these requirements include:

(a) A fully featured programming language binding at the application programming interface to enable many key services and to provide the application framework.
(b) A language binding allowing application programs to interact with core system services using portable techniques.
(c) Facilities that provide some form of local interprocess communication (IPC) and data sharing for intra-platform application software entities.

4. **Information Interchange Services**

The Information Interchange Services defines the services across which external, persistent storage media is provided, where only format and syntax is required to be specified for data portability and interoperability. As previously mentioned, there are no specifications associated with this profile from this section because at this time there has been no specific instances for using them. Some of these requirements include:

(a) Filesystem access to persistent storage using portable methods and techniques.

5. **Miscellaneous Services**

The Miscellaneous Services as defined in this report do not map directly into the OSE-RM model. They are outside the scope of the OSE-RM and further define the requirements needed to support parallel and supercomputing aspects in a distributed setting. In the future, these issues will need to be addressed in order to provide

these services to the general purpose distributed system framework. Some of these requirements include:

(a) Library support for differences in supercomputing architectures.

(b) Common ways for applications to express a particular parallel programming paradigm and a means to interject multiple types of paradigms into the application.

(c) Distributed shared memory in order to emulate the shared memory multiprocessing architecture paradigm.

(d) Message passing capabilities in order to emulate the multiprocessing architecture paradigm.

(e) Support for multiple parallel computational models, high-level communication routines that provide broadcast, barrier synchronization, global extrema finding, and tracing capabilities.

(f) Fault tolerant mechanisms, providing restart, migration and checkpointing of application instances.

(g) Provisions for security services throughout all aspects of the distributed system.

(h) Tools that provide distributed system management and administration.

5 Conclusion

In the parallel and supercomputing areas, PVM represents a powerful distributed system infrastructure for achieving coarse-grained, loosely coupled heterogeneous computing. It is especially attractive due to its availability in the public domain. Our goal for using PVM was to extract as many requirements issues from the parallel and supercomputing application domains as we could and apply those to the OSE-based distributed environment. The requirements gathering process was based on researching, installing, building, developing and executing application programs for the PVM system. By initiating *hands-on* activities with PVM, an adequate requirements framework needed to support the parallel and supercomputing domains have been captured. The results of our study include a collection of core requirements that assist in the definition of the NIST DPP. The core requirements have been partitioned into service areas representing distributed system aspects from the OSE reference model definition. Mapping available standards and specifications to the core requirements found from this and other application domain studies will be initiated as part of a final DPP activity. As a research by-product, this report represents an appropriate reference point for users' who are examining these types of software packages for use in their specific domain.

References

[1] Birrell, D. A., Nelson, B. J., *Implementing Remote Procedure Calls* ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, Pages 39-59.

[2] Geist, G. A., Sunderam, V. S., *Network Based Concurrent Computing on the PVM System* Technical Report, Oak Ridge National Laboratory, 1990.

[3] Geist, G. A., Sunderam, V. S., et. al., *PVM User's Guide and Reference Manual* Technical Manual 12187, Oak Ridge National Laboratory, May 1993.

[4] Hwang, K., Briggs, F. A., *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

[5] Leffler, H., et al. *The Design and Implementation of the 4.3 BSD UNIX Operating System.*, Addison Wesley, New York, 1988.

[6] Sunderam, V., *PVM: A Framework for Parallel Distributed Computing*, Concurrency: Practice and Experience Vol. 2 No. 4, Dec. 1990.

[7] Marshak, David, S., *ANSA: A Model for Distributed Computing*, Network Monitor: Guide to Distributed Computing, Vol. 6 No. 11, Nov. 1991.

Acknowledgements

Thanks to Christine Piatko and David Su for their important comments during the manuscript review process. The author greatly acknowledges the researchers at the Oak Ridge National Laboratory and Emory University for developing PVM, and subsequently placing it in the public domain. The PVM architecture image was collected from the Internet site that maintains the PVM source online. The HeNCE GUI image was generated internally here at NIST. Portions of this research were funded by NIST Scientific and Technical Research Services (STRS) and in part by the Department of Defense (DoD), Defense Information Systems Agency (DISA) Heterogeneous Distributed Systems Environment (HDSE) program of the Office of the Secretary of Defense. The PVM source code, documentation, and research articles can be obtained from the University of Tennessee Internet site *netlib2.cs.utk.edu*, corresponding to the Internet address 128.169.92.17, for non-naming sites. ♣