PVM : A Framework for Parallel Distributed Computing*

V. S. Sunderam

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

ABSTRACT

The **PVM** system is a programming environment for the development and execution of large concurrent or parallel applications that consist of many interacting, but relatively independent, components. It is intended to operate on a collection of heterogeneous computing elements interconnected by one or more networks. The participating processors may be scalar machines, multiprocessors, or special-purpose computers, enabling application components to execute on the architecture most appropriate to the algorithm. **PVM** provides a straightforward and general interface that permits the description of various types of algorithms (and their interactions), while the underlying infrastructure permits the execution of applications on a virtual computing environment that supports multiple parallel computation models. **PVM** contains facilities for concurrent, sequential, or conditional execution of application components, is portable to a variety of architectures, and supports certain forms of error detection and recovery.

1. Introduction

In recent years, parallel and distributed processing have been conjectured to be the most promising solution to the computing requirements of the future. Significant advances in parallel algorithms and architectures have demonstrated the potential for applying concurrent computation techniques to a wide variety of problems. However, most of the research efforts have concentrated either upon *computational models* [1], parallel versions of *algorithms*, or machine *architectures*; relatively little attention has been given to software development environments or program construction techniques that are required in order to translate algorithms into operational programs. This aspect is becoming more important as parallel processing progresses from the solution of stand-alone, mathematically precise, problems to larger and more complex software systems. Such systems often consist of many interacting components, each with its unique

* 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.

requirements. These requirements are often diverse enough that it is difficult to program them within a unified framework, and cumbersome, and sometime impractical, to execute the system on a single machine.

An example of such a system is the fluid dynamics application termed BF3D, described in [17]. This application is not only compute intensive; it uses large amounts of memory, creates very large quantities of output data, and requires 2D and 3D graphics terminals to display results. The algorithm essentially involves the solution of a set of partial differential equations in three dimensional space, by using an appropriately staggered grid. The algorithm may be parallelized in different ways, enabling its execution on several categories of parallel machines. However, typical multiprocessors rarely support high-bandwidth external I/O or high performance graphics, thereby necessitating the separation of the computation, output management, and graphical display components of the system. Another example of an application with differing requirements for individual sub-algorithms is the Global Environment Simulation project [2], a large simulation effort to study contaminant concentrations and dispersal characteristics as a function of various environmental factors. The ideal computational requirements of this simulation are vector processing (for fluid flow analysis), distributed multiprocessing (modeling contaminant transport), high-speed scalar computation (simulation of temperature effects), and real-time graphics for user interaction.

From a different point of view, many applications that are amenable to concurrent execution can be programmed using either message passing or shared memory algorithms; when computing environments supporting both are available, a hybrid algorithm, at a sufficiently high level of granularity, may be used to advantage. A straightforward, but illustrative example is matrix multiplication using subblock decompositions [18]. In an environment consisting of different types of scalar machines and multiprocessors, highly effective and efficient algorithms for matrix multiplication can be implemented, with some subblocks being multiplied using static prescheduling on shared memory machines and others using "pipe-multiply-roll" strategies on distributed memory computers. Empirical results obtained by using this approach are presented in Section 5.

It should be noted that most typical computing environments already possess the hardware diversity required to solve such large, parallel applications, and also contain support for multiple concurrent computation models. High speed local networks with graphics workstations, high-performance scalar engines, an occasional multiprocessor, and perhaps a vector computer are the norm rather than the exception, and will continue to be over the next few years. However, to harness this collection of capabilities and to utilize it productively requires considerable efforts in coordination and reconciliation between different computation models and architectures — all of which has to be done manually. The **PVM** (Parallel Virtual Machine) project is an attempt to provide a unified framework within which large parallel systems can be developed in a straightforward and efficient manner. The overall objective of this project is to permit a collection of heterogeneous machines on a network to be viewed as a general purpose concurrent computation resource. Application algorithms are expressed using the most suitable paradigm; the **PVM** system executes them on the most appropriate hardware available, either directly or by emulating the particular computation model. Furthermore, it is frequently desired to incorporate existing software (preferably with little or no modifications) into a larger system; the **PVM** system is designed to enable this in a convenient and natural manner.

The **PVM** system provides a set of user interface primitives that may be incorporated into existing procedural languages. Primitives exist for the invocation of processes, message transmission and reception, broadcasting, synchronization via barriers, mutual exclusion, and shared memory. Processes may be initiated synchronously or asynchronously, and may be conditioned upon the initiation or termination of another process, or upon the availability of data values. Message transmission as well as file output may be preceded by invocations of specially provided primitives to ensure that data is transmitted or stored in a machine independent form. Application systems may be programmed using these primitives in the language of choice; different components may even be programmed in different languages. The **PVM** constructs therefore permit the most appropriate programming paradigm and language to be used for each individual component of a parallel system, while retaining the ability for components to interact and cooperate. It should be mentioned that the **PVM** user-interface primitives have been partly derived from and are a superset of the portable programming constructs described in [3]; an application written using these primitives may therefore also execute directly on a specific multiprocessor when necessary.

The **PVM** system consists of support software that executes on participating hosts on a network; the network may be local, wide-area or a combination, and the host pool may be varied dynamically. Hosts may be scalar machines, workstations, or parallel processors — the latter being considered an atomic computational resource by **PVM**. This support software interprets requests generated by the user-level constructs and carries out the necessary actions in a machine independent manner. The **PVM** software essentially consists of a collection of protocol algorithms to implement reliable and sequenced data transfer, distributed consensus, and mutual exclusion. In an attempt to make the system as robust as possible, these algorithms also incorporate error detection mechanisms — process and processor failure notification is provided to applications, thereby enabling them to take corrective action. In addition, heuristics are included into the algorithms for mutual exclusion and consensus that enable the preemption of deadlock in several situations.

Several projects similar to **PVM** have been undertaken in the past, and some are ongoing. A few representative examples are listed below, with comparisons to **PVM**. The DPUP library [4] emulates a loosely coupled multiprocessor on a local network, as does the dsim [5] system and the Cosmic environment [6]. The two latter systems require the preconfiguration of a virtual machine on which applications execute and support only basic message passing mechanisms. The Amber project [15] is somewhat different in that the targeted environment is a collection of homogeneous *multi*-processors. One of the operating modes within DPUP, as well as projects such as Marionette [7] and MSPCM [8], uses the master-slave approach, where a central controlling process is responsible for, or is involved in, every system event. In addition to affecting performance and being an unnatural model for certain classes of problems, this central process is critical, and its failure leads to a complete collapse of the entire system. Another shortcoming common to all the above is the use of virtual circuits for network communication; in addition to overheads that may not be justifiable, practical limits on the number of connections affect the scalability of applications. In addition, failure resiliency and debugging support is minimal. The **PVM** system is completely distributed, supports a dynamic host pool, and assumes only that an unreliable, unsequenced datagram delivery mechanism is available. From the application's point of view, **PVM** constructs are substantially more general in nature and encompass both the message passing and shared memory paradigms; yet, by substituting alternative libraries, unmodified programs may execute on specific multiprocessors.

The following section describes the user interface and the important design aspects of **PVM**. An overview of the **PVM** support software, with an emphasis on the protocol algorithms and key implementation features follows. Preliminary results and performance figures are then presented, and the concluding section reports on continuing and future work.

2. The User Interface

The application views the **PVM** system as a general and flexible parallel computation resource that supports common parallel programming paradigms. Application programs access these resources by invoking function calls from within common procedural languages such as C or Fortran. Such an interface was selected primarily for portability reasons — most multiprocessor applications are currently written in procedural languages with embedded, machine-specific, function calls that perform process spawning, message reception and transmission, and shared memory operations. The **PVM** primitives have been made the same as or very similar to the union of these functions, thereby enabling previously written applications to be ported readily to the **PVM** environment. This aspect also permits the execution of applications, or components thereof, on specific machines when possible. The **PVM** user interface syntax and semantics are presented in this section with illustrative examples using the C language interface.

2.1. Processes and Process Initiation

In the **PVM** system, an application is considered to consist of *components*. For example, a simulation application might consist of a partial differential equation component, a matrix solution component, and a user interface component. It should be pointed out that this definition of a component is perhaps unconventional; usually, the term implies a phase or portion of an application that is embodied in a subroutine – such as "the forward-substitution component of a matrix solver". However, the **PVM** system is a large-granularity environment, primarily targeted at applications that are collections of relatively independent programs. In view of this, a **PVM** component corresponds not to a phase in the traditional sense, but rather to a larger unit of an application. From the system point of view, a component corresponds to an object file that is capable of being executed as a user-level process. A compiled C program that performs LU factorization is an example of a component. It is the responsibility of the user to compile component programs to all target architectures on which that component may execute. Depending upon the target machine, the compiled version of a component may either link against the **PVM** primitives, or machine specific libraries, or both. A component is therefore a static entity and is identified by a *name*; associations between component names and executable versions are set up as discussed in the following paragraphs.

A complete description of application components, i.e. the component name and all corresponding executables (each with an architecture tag), is obtained by the **PVM** support software. This information is gathered either from a file or from a startup process, as will be explained below. An example of a component description file is shown in Figure 1. This table illustrates that a component, identified by a name, may be manifested as several different executable files; and conversely, that multiple component names may map onto the same executable. The first feature permits the **PVM** system to execute components at the most suitable location, while the second allows the user to specify a particular location as will be explained below.

Name	Location	Object file	Architecture
factor	iPSC	/u0/host/factor1	ipsc
factor	msrsun	/usr/alg/math/factor	sun3
factor	msrsun	/usr/alg4/math/factor	sun4
chol	csvax2	/usr/matrix/chol	vax
chol	vmsvax	JOE:CHOL.OBJ	vms
tool	msrsun	/usr/view/graph/obj	sun3
factor2	iPSC	/u0/host/factor1	ipsc

Figure 1 : Example Component Description File

A *process* is an executing instance of a component and is identified by the component name and a positive instance number. Processes may be initiated from within components or from a "startup" process that may be manually executed on any participating host. A process is initiated by invoking the **initiate** primitive with the component name as an argument; the instance number of the initiated process is returned to the user. Prior to executing any **PVM** construct, however, a process must invoke the **enroll** function; this establishes a (machine dependent) mechanism by which a user process may communicate with the **PVM** system. A typical section of code executed by a startup process is shown in Figure 2. It should be noted in the example shown that the physical location of the initiated processes is transparent to the invoking process; the **PVM** system determines the best machine on which to execute a process based upon the current host pool, the alternative architectures on which a component may execute, and the load factor on those machines. However, a specific location may be forced by declaring a new component name (as in the last line of the component description file above) and initiating that component.

```
...
enroll("startup");
for (i=0;i<10;i++)
   instance[i] = initiate("factor");
...
```

Figure 2 : Initiation of multiple component instances

Positioning of application components on specific processing elements is often a desirable facility, particularly in heterogeneous environments. When the application has prior knowledge of the special capabilities of a specific machine or even the characteristics of a particular dataset, executing a component at a fixed location can lead to significant benefits. In **PVM**, this may be accomplished either statically as explained above, or dynamically by using the **entercomp** construct. Arguments to this construct are a component name, an object filename and location, and an architecture tag. This construct is used to "register" a new component that may subsequently be executed by using the **initiate** construct. For example, if an executing component found it necessary to spawn a subprocess on a specific machine, it might use the following section of code:

```
entercomp("subproc","sequent8","/usr/a.out","seq");
initiate("subproc");
```

The **initiate** mechanism is, by default, asynchronous. Control is returned to the invoking process as soon as the instance number of the process is available. However, under certain circumstances, it may be necessary to initiate a component only after another process has terminated. The **initiateP** variant allows this by permitting the user to defer initiation of a component until after another has terminated. For example,

```
initiateP("factor","matmul",3);
```

will initiate an instance of "factor" only after instance number 3 of "matmul" has terminated. A third-argument value of 0 will cause "factor" to be initiated only when *all* instances of "matmul" terminate. In an analogous fashion, **initiateD** is used to execute components conditional upon the occurrence of a user-signaled event, normally the availability of data. Thus,

```
initiateD("chol","dataset7");
```

will delay the execution of "chol" until some other process signals the occurrence of the "dataset7" event, by invoking the **ready**("dataset7") primitive. All variants of **initiate** return a negative result if a process could not be initiated, thereby enabling the invoker to take appropriate action. The global component dependencies of the application may therefore be specified within a startup process by the use of appropriate **initiate** primitives or variants, embedded within common selection and iteration control flow constructs available in the host language. Of course, a component itself may be composed of several subcomponents — whose dependencies and execution order are indicated in an analogous manner within that component. Two other constructs termed **terminate** and **waitprocess** are also provided. Both take a component name and an instance number (or 0 to mean all instances) as arguments; the first aborts the process while the second blocks the caller until the process completes.

2.2. Data Transfer and Barrier Synchronization

Inter-process communication via message passing is one of the basic facilities supported by **PVM**. In the interest of portability and wide applicability, the primitives to accomplish message transfer have been derived from existing implementations (e.g [9]), including those described in [3]. Certain aspects, however, are necessarily different; primary among them is addressing. Since the physical location of processes is deliberately transparent to user programs, message destinations are identified by a {component name, instance number} pair. Furthermore, owing to the heterogeneous nature of the underlying hardware that **PVM** executes upon, it is necessary for user programs to send and receive typed data in a machine independent form. To enable this, a set of conversion routines has been provided — user programs invoke these routines to construct message buffers and to retrieve data values from incoming messages.

In keeping with popular message passing mechanisms, the **PVM** **send** and **receive** constructs incorporate a "type" argument. This is the only argument to **receive**, while **send** requires a destination component name and instance number as additional arguments. The type parameter permits the selective reception of messages and has been found to be extremely useful in practical applications. It should be noted that neither the data buffer itself nor its length appear explicitly as arguments — owing to data representation and size differences on different machines, user programs should only access messages using the conversion routines. Shown in Figure 3 is an example of data transfer between two component processes.

```
/* Sending Process */
/*-----------------*/
initsend();                        /* Initialize send buffer */
putstring("The square root of "); /* Store values in        */
putint(2);                         /* machine independent    */
putstring("is ");                  /* form                   */
putfloat(1.414);
send("receiver",4,99);             /* Instance 4; type 99    */

/* Receiving Process */
/*-------------------*/
char msg1[32],msg2[4];
int num; float sqnum;
recv(99);                          /* Receive msg of type 99 */
getstring(msg1);                   /* Extract values in      */
getint(&num);                      /* a machine specific     */
getstring(msg2);                   /* manner                 */
getfloat(&sqnum);
```

Figure 3 : User process data transfer

In order for a receiving process to obtain additional information about the most recently received message, the **recvinfo** construct is provided; this returns the name and instance number of the sending process and the message length. In addition, two variants of the **recv** construct are provided. The first, **recv1**, permits the user to specify the maximum number of messages of other types that may arrive in the interim (i.e. while waiting for a message of the specified type). If a message of the anticipated type does not arrive within this window, an error value is returned to the program, thus enabling the detection of and possible recovery from incorrect program behavior or unacceptable levels of asynchrony. The second variant, **recv2**, allows the specification of a timeout value and is valuable in preventing certain forms of deadlock as well as in user-level detection of failed components. Also provided is the **broadcast** primitive that sends a message to all instances of a specified component.

Synchronization via barriers is a common operation in many applications. Under **PVM**, barrier synchronization is accomplished using the **barrier** construct. This construct takes an integer argument; an instance of a component invoking this construct will block until the specified number of instances of the component also arrive at the barrier. For example, to measure the performance of algorithm sections, participating components may execute the following section of code:

```
. . .
barrier(k);       /* There are k instances executing */
/* Get start time */
/* Perform computations */
/* Get end time */
barrier(k);
/* Compute global maximum of (end time - start time) */
```

When the barrier construct has the above semantics however, it is possible that different instances invoke the **barrier** construct with different arguments. Any such conflicts are resolved by using the maximum specified number as the barrier count. The **PVM** system also attempts to detect and correct barrier deadlocks by notifying invoking processes if some instances of a component terminate before they reach a barrier, and the residual number of instances cannot form a barrier quorum. In such situations, live processes return from a **barrier** call with a negative result value. In addition to barriers, or as an alternative, the **waituntil** construct is also provided as a means of synchronization. This construct (suggested in [10]) takes an event name as an argument and blocks until another process indicates the occurrence of that event by using the **ready** primitive mentioned earlier.

2.3. Shared Memory and Mutual Exclusion

The use of shared memory to synchronize and communicate between processes is a convenient and well understood paradigm, and the **PVM** system provides such an interface for algorithms that are best expressed in these terms. This facility, however, should be used judiciously — since shared memory is emulated by **PVM** on distributed memory architectures, some performance degradation is inevitable when the granularity of access is very fine. Nevertheless, such a feature is useful and valuable. For example, when using the "bag of tasks" and "worker pool" approach, the shared memory model permits greater control, increased overall throughput, and is affected less by load imbalances, provided the tasks are of sufficiently large size. Since **PVM** permits each worker in the pool to run on different architectures, the individual worker components (or groups thereof) may be written to *internally* use either message passing or shared memory. Section 5 describes an experiment where this facility was successfully used to enable a shared memory multiprocessor to cooperate with a hypercube in the solution of an application problem.

The shared memory primitives provided are modeled once again after popular, existing implementations. A shared memory segment is first allocated by invoking the **shmget** construct that takes a string valued identifier and a segment size in bytes. To acquire a shared memory segment for use, a user process invokes the **shmat** construct, specifying the segment identifier, the address within the process at which the segment is to be mapped, a flag indicating whether the segment is to be mapped read-only or read-write, and a timeout value. This construct implicitly incorporates a lock operation; if mutually exclusive access to the segment cannot be provided, the invoking process is suspended — for a period not to exceed the specified timeout value.

The attach operation described above maps a contiguous, untyped block of bytes at the specified address. In most situations however, shared memory segments will be used to store and manipulate typed data. In order to permit this among dissimilar machines, typed variants of the attach construct are provided. For example, the **shmatint** construct takes an integer pointer as its second argument, while the **shmatfloat** variant is used for shared memory regions that hold floating point values. (It should be noted that typed data transfer between dissimilar architectures could lead to loss of precision or to truncation owing to wordsize differences. Both message passing and shared memory mechanisms are subject to this drawback. The PVM system attempts to minimize this by utilizing the largest size possible for typed data values.) When a process no longer needs exclusive access to a region, it invokes the **shmdt** construct (or a typed variant) whereupon the lock is released and the region unmapped. Finally, the **shmfree** construct is used to deallocate a segment of shared memory when it is no longer required. Shown in Figure 4 is an example of the use of these constructs to pass an array of real numbers between two processes.

```
/* Process A */
/*-----------*/
if (shmget("matrx",1024)) error();/* Allocation failure    */
while                            /* Try to lock & map seg */
   (shmatfloat("matrx",fp,"RW",5));
for (i=0;i<256;i++) *fp++ = a[i]; /* Fill in shmem segment */
shmdtfloat("matrx");              /* Unlock & unmap region */

/* Process B */
/*-----------*/
while                             /* Lock & map; note:reader*/
   (shmatfloat("matrx",fp,"R",5));/* may lock before writer */
for (i=0;i<256;i++) a[i] = *fp++; /* Read out values        */
shmdtfloat("matrx");              /* Unlock & unmap region  */
shmfree("matrx");                 /* Deallocate mem segment */
```

Figure 4 : Use of shared memory for IPC

While shared memory is perhaps the most common resource that processes require mutually exclusive access to, it is possible that the PVM environment contains other resources that processes must access in a similar manner. To accommodate such requirements, a generalized locking facility is also provided. The **lock** construct permits the logical locking of an entity that is named by a string argument; the PVM system blocks other processes wishing to lock this entity until the possessor invokes the **unlock** construct. For example, different components of a large application may wish to output results periodically to a user terminal. To avoid interference and to distinguish the source of the output, components may adopt a convention that requires locking "`terminal`" before printing messages or results. Another situation where such a facility could be useful may be found in the shared memory example in Figure 4. In that example, it is easy to see that the processes may access the shared memory segment in an incorrect *order* even though each will have exclusive access to it. A possible rectification of this situation is to use the **lock** construct as shown in Figure 5; however, in practice it is more likely that a transmitted message or the **waituntil** facility will be used to resolve such situations.

```
/* Process A */
/*----------*/
lock("fillmatrix",5);
/* Allocate, attach, fill, and detach shared mem segment. */
unlock("fillmatrix");

/* Process B */
/*----------*/
loop:
   lock("fillmatrix",5);
   if (shmatfloat(...) == SEGMENT_NONEXISTENT) {
      unlock("fillmatrix");
      sleep(1);
      goto loop;
   }
/* Read values out of shared mem segment, detach, & free */
```

Figure 5 : Use of the `lock` construct

It should be noted that the **lock** construct is intended in **PVM** for obtaining exclusive access to resources at a large-grained level, as opposed to loop level locking of shared variables that is common on shared memory machines. Once again, a component that executes on such a machine would be programmed to use internal locking constructs where appropriate, and **PVM** locking facilities for resource sharing between other components that executed on other architectures. An example code skeleton depicting this situation on a Sequent shared memory multiprocessor [19] is shown in Figure 6.

2.4. Miscellaneous Facilities

In addition to the primary constructs described in the preceding sections, a few miscellaneous constructs are also provided. The **status** construct takes a component name and instance number as arguments and returns status and location information regarding that component. The **entercomp** construct permits dynamic additions to the component description table. The **shmstat** construct is used to obtain information about active shared memory regions, while the **lockstat** primitive reports the status of active locks. A complete list of all the user interface constructs

```
        . . .
        CALL shmatfloat("A_row1",A,"RW",5)
        {Lock & map row 1 of an external shared array to A}
        . . .
C$DOACROSS SHARE(A,index)
        do 20 i = 1, 100
        . . .
        CALL m_lock
        {Local, loop level lock to indicate critical section}
        index = index + 1
        CALL m_unlock
        . . .
20      continue
        . . .
        CALL shmdtfloat("A_row1")
        {Unlock and unmap external shared array}
        . . .
```

Figure 6: **PVM** locking contrasted to machine dependent locking

along with their argument lists and a one-line description of each is given in the appendix.

3. PVM System Design and Implementation

The **PVM** support software executes as a user-level process on each host in the participant pool. An initial set of participating hosts is statically identified; additions or deletions are possible during operation by means of an administration interface. The **PVM** system is designed to be implemented in a manner that requires no operating system changes or modifications, and porting efforts to varied operating system environments are minimal. The **PVM** support process (termed *pvmd*) on a host is responsible for all application component processes executing on that host; however, there is no central or master instance of *pvmd*. Control is completely distributed (by virtue of all *pvmd* processes possessing global knowledge) in the interest of avoiding performance bottlenecks and increasing fault tolerance. The *pvmd* processes are initiated on each participating host either manually, through the administration interface, or via a machine/OS dependent mechanism such as *inetd* in the Unix environment. In this section, the key design aspects of the *pvmd* software are discussed with an emphasis on the protocol algorithms used.

3.1. Basic Facilities

In terms of network capabilities, the **PVM** system assumes only that unreliable, unsequenced, point-to-point data transfer (but with data integrity) facilities are supported by the hardware platform on which it executes. The required reliability and sequencing, as well as other necessary operations such as broadcast, are built into the **PVM** system in the interest of efficiency and portability. While it is true that most operating systems in existence already support reliable and sequenced data delivery, in most cases this is via the use of virtual circuits — for the projected use of **PVM**, the overheads and scalability limitations of using such a service directly did not warrant its adoption. In the test implementations of **PVM**, the UDP [11] protocol was used; this deliberate choice of a simple datagram protocol also permits relatively simple porting or protocol conversion when **PVM** is to be installed under a different operating system environment.

Across the network, *pvmd* processes communicate using UDP datagrams. The "well known port" approach is used for addressing; all incoming messages are received by *pvmd* processes on a predetermined port number. For user-process to user-process communication, the following scheme is employed. The first communication instance between any two entities is routed through the *pvmd* processes on the source and destination machines. Location and port number information is appended to this exchange; the **PVM** routines (linked to the user process) that implement **send** and **recv** cache this information, thus enabling direct communication for subsequent exchanges. Local user processes communicate with *pvmd* using the most efficient machine dependent mechanism available and the development of this mechanism is deemed part of the installation procedure. However, the generic version of *pvmd* may be adopted; this utilizes UDP datagrams once again, via the loopback interface if one is available. The *pvmd* process uses a different, predetermined port number for incoming messages from all local user processes.

3.2. Point-to-Point Data Transfer

To achieve reliable and sequenced point-to-point communications, the *pvmd* processes use a positive acknowledgment scheme and an additional header that contains sequence numbers as well as fragmentation and reassembly information. Unacknowledged transmissions are retried a parameterized number of times after which the recipient process or processor is presumed to be inoperative. The sequence numbers are destination specific and are used by the message recipient for sequencing as well as for duplicate detection. The header is placed at the end of a UDP datagram to reduce copying overheads, and single datagram sizes are restricted to the smallest MTU (maximum transmission unit) of all participating hosts. When first initiated, *pvmd* processes determine the protocol specific addresses of all participating hosts and proceed to service incoming requests from the network or user processes in an infinite loop.

Each *pvmd* process maintains information concerning the location and status of all application component processes. A user **send** is addressed to a component name and instance number; the local *pvmd* determines the physical location of that process and forwards the message to the remote *pvmd*. As described, the user process library performs this translation for the second and subsequent messages. The source component name and instance number are appended to the message, enabling message delivery with an indication of the sender’s identity. As mentioned, executing the **enroll** construct is a precondition to user process participation — in the UDP implementation, this supplies to the local *pvmd* the receiving port number of the user process.

3.3. Message Broadcasting

Broadcast is a commonly performed operation in the **PVM** system, both because applications desire such a facility and since it is inherent to the completely distributed nature of the **PVM** support software. All *pvmd* processes maintain information regarding all processes, shared memory segments, and locks, to guard against loss of context and state in the event of failures. User process broadcasts are first delivered sequentially to local recipient user processes after which the local *pvmd* process broadcasts over the network to all other *pvmd* processes that in turn, deliver the message to their local user process recipients. Although most computing environments support a network broadcast facility, *pvmd* broadcast is implemented in **PVM** using point-to-point messages with recursive doubling. This decision was made in the interest of portability and efficiency; given that network broadcast is unreliable, acknowledgments are necessary from each recipient, resulting in $O(p)$ time ($p + 1$ sequential steps are to be performed by the originator), while recursive doubling broadcast is accomplished in $O(\log_2 p)$ time, where p is the number of processors. The participating pool of hosts is logically numbered from 0 to $p-1$, and the originator (or root) of the broadcast is part of the broadcast message. There is one *pvmd* process per host, which represents that processor. Broadcast proceeds in "rounds", with the number of processors contributing to the broadcast effort doubling in each round. In any round, processor i transmits to $i + 2^r \; mod \; p$ and receives an acknowledgment. A processor j joins the broadcast effort at round r_j, where $r_j = no.\ of\ significant\ bits\ in\ (j - root)\ mod\ p$. In the event of processor failure, the *pvmd* process that first detects the failure assumes the broadcast duties of the failed processor. If the quantity $2(2^r)$ is less than $2 \log_2 p$, failure notification is piggybacked on the broadcast, at the end of which the remaining processors are individually informed. Otherwise, the detecting processor initiates another broadcast with failure information, at completion of the current broadcast.

The *pvmd* processes execute a finite state machine which gives precedence to messages (requests) incoming while another activity is in progress. Such a scheme is adopted to avoid deadlock; two processes transmitting to each other may both wait indefinitely for each others acknowledgment if this precedence rule were not followed. It should also be pointed out that in the case of **PVM** hardware platforms where wide area networks are involved, the choice of an appropriate timeout value can significantly affect the performance of the data transfer mechanisms and the broadcast process. Further, the present implementation does not perform any optimizations in the broadcast scheme when a geographically distant host is at a non-leaf position in the broadcast spanning tree.

3.4. Mutual Exclusion

Mutual exclusion is another primitive required both in response to user requests as well as for *pvmd* coordination. Examples are exclusive access to emulated shared memory, general resource locking, and assignment of unique instance numbers for application component processes. Distributed mutual exclusion is normally achieved by unanimous or majority consensus; a requesting process that receives permission from a certain number of processes is deemed to have acquired the lock. Different strategies, varying in their approach, efficiency, and level of failure resiliency have been proposed and representative methods are described in [12, 13, 14]. The strategy adopted in **PVM** is somewhat different from these approaches, but the algorithm is efficient and, more importantly, is integrated with the required distribution to all *pvmd*s of lock location information.

A *pvmd* process, either for its own purposes or on behalf of a local user process, attempts to obtain a lock by broadcasting a "claim" for the lock. Since all *pvmd* processes possess knowledge regarding the use (and location) of all locks, such an attempt will, of course, only be made when a lock is known to be free at the start of the claim. In the absence of conflicting claims (a situation most likely to be encountered in practice), the requester, after the broadcast has been completed, can assume that the lock has been successfully obtained. In the process, all other *pvmd*s update their lock table information, and (implicitly) grant the requester permission for exclusive access to the particular resource.

It is of course possible that two processes may initiate claims on the same resource before either has received the other's request. In **PVM**, such situations are resolved using a heuristic that assumes that communication between any pair of processors takes the same amount of time. In particular, consider two processors (*pvmd* processes) A and B that wish to acquire the same lock, and another processor C. Note that $\{B,C\}$ and $\{A,C\}$ are in the broadcast spanning trees of A and B respectively, possibly at different depths. It may be assumed without loss of generality that A's processor number is less than that of B. Under the constant time assumption, both A and B will receive each others claims before their broadcast is completed. When B receives A's claim, it computes the number of rounds that A's broadcast has proceeded; if this number is greater than the number of rounds that its own broadcast has proceeded, B surrenders its claim to the lock. An identical (first-claim, first served) policy is followed by A. If the broadcast progress metrics are the same, the lower numbered processor is given priority and is considered to have obtained the lock. The passive processor C also makes the same decision since it has the capability of computing the number of rounds of broadcast progress that each claimant has made when the second such broadcast arrives.

In practice however, communication times between arbitrary pairs of processors may not be constant; further, intervening messages of other types may skew the propagation time of a broadcast claim. For practical safety therefore, the originators of conflicting claims exchange a confirmatory message — with the claim being abandoned if their respective notions of the successful claimant are not in agreement. In such a situation, the lower numbered processor broadcasts a "reset lock" message, and the entire process is started afresh, but without competition from the "losing" processor. In case of process or processor failures, the strategy of all *pvmd* processes possessing all information is used to avoid undesirable situations. If a process or processor holding a lock terminates without releasing it, the particular resource is marked as "defunct"; further requests to the resource are denied until an explicit reset is performed.

3.5. Process control

The initiation order and process dependencies of application components are described by the use of appropriate **initiate** constructs embedded within host language control flow statements as described in the preceding section. This implies that it is not possible to determine statically the application process flow graph as component initiations may be conditional or repeated based upon parameters known only at execution time. The **PVM** system therefore performs process initiations in response to requests based upon the resources available and load conditions at the moment of the request — rather than by constructing a predetermined static schedule and process to processor assignments.

When an application component process makes an **initiate** request, the local *pvmd* process first determines a candidate pool of target hosts based upon the information in the component description file. One host is then selected from this pool based upon the following algorithm:

(1) Select next host from pool in round-robin manner, based upon all initiations that originated here.

(2) Obtain load metric (decayed average of number of processes in run queue) from this potential target host.

(3) If this quantity is less than a prespecified threshold, select this host.

(4) Otherwise, repeat the process. If no host has a load factor below the threshold, the host with the lowest load is the selected target.

Once the target host is identified, the local *pvmd* sends the initiate request to the *pvmd* process on the remote host, where the application component is initiated. The remote *pvmd* then broadcasts notification of this event to all processors, simultaneously claiming an instance number for this initiation (by simply incrementing the last previous instance number for the component). Conflicting claims for the same instance number are again resolved as in the case of multiple claims to a lock, with a "losing" processor using a higher value. Once again, consistent conflict resolution is confirmed by an exchange of messages between all claimants and reset actions are performed in the case of disagreement. Application process termination information is also broadcast to all *pvmd* processes. Conditional variants of **initiate** are saved by the local *pvmd*, and this queue is inspected and appropriate action taken when the particular event occurs.

3.6. Shared Memory and Barriers

In the **PVM** system, shared memory is emulated by first creating an image of a memory segment on secondary storage. A file of the requested size is created; for efficiency and failure resiliency reasons, the local *pvmd* (the processor at which the creation request originated) attempts to locate the file on a device that is accessible to other processors via a network file system. Mutual exclusion, both for creation as well as for access, is achieved as described earlier. A *pvmd* process that has acquired a lock (on behalf of a local application process) copies the file into the requested address space; this is done directly if the file is accessible directly, and with the assistance of the remote *pvmd* if it is not. A user release request results in the specified memory area being copied back to the file unless the lock request was for read-only access. It should be noted that creation, locking, unlocking, and deallocation (resulting in file removal) events are broadcast to all *pvmd*s; given the conflict resolution rules and highest priority to incoming requests, undesirable inconsistencies are avoided.

Barrier synchronization in **PVM** is accomplished by using an efficient algorithm described in [20]. The algorithm considers the *pvmd* processes as being the vertices in a logical quadratic network. In such a network each vertex i has outgoing links to j such that $j = i + 2^r \; modN$, where N is the number of *pvmd* processes, and $0 \le r < \log_2 N$. Upon receipt of a **barrier** request from one of the local application processes, each *pvmd* process i first waits until all local components have also reached the barrier, and then executes the following algorithm:

for r := 0 to $\log_2 N - 1$ do
    send message containing r to $i + 2^r \; mod \; N$
    wait for receipt of message containing r

This algorithm is efficient both in the number of messages used and the number of rounds, and may be used for arbitrary values of N.

3.7. Discussion

An overview of the design and implementation strategies used to realize the various **PVM** features has been presented in this section. As is common with most software systems, there is sometimes a tradeoff between versatility and efficiency. In **PVM**, the shared memory facility that is emulated on a network is likely to be the cause of performance degradation if used unwisely. However, as has been pointed out, when the grainsize of sharing memory across the distributed environment is relatively large, acceptable efficiency levels may be obtained; and the benefits obtained by the shared memory abstraction outweighs the small performance degradation. It should be noted that mutual exclusion and barrier synchronization are implemented independent of the shared memory mechanism and use algorithms that are known to be efficient for tens of processors on local networks.

4. Preliminary Results

To facilitate its use and to determine its effectiveness, the **PVM** system has been implemented on a variety of machines including Sun 3/50, 3/60 and SparcStation1 workstations, Vax 11/785 and Sun 4/280 servers, a 64 node Intel iPSC/2 hypercube, and a 12 processor Sequent shared-memory multiprocessor. The minimal assumptions made regarding the underlying facilities available greatly simplified the implementation efforts; the software could be ported (from a base Sun 3 implementation) to to all the environments with changes necessary only in data representation and conversion utilities. In the implementation, the *pvmd* processes run independently, while the user level routines are supplied as a set of libraries to be linked in with application components. This section reports on observed performance figures for the basic **PVM** primitives and two straightforward applications; in the next section, two other applications that make extensive use of the heterogeneous facilities are described.

4.1. Basic Primitives

The efficiency of user-level data transfer is perhaps the most critical aspect of any distributed computing environment. In the **PVM** system, low latency data transfer has been provided without sacrificing location transparency, and a datagram protocol is used so that overheads of more heavyweight protocols are avoided. It is anticipated that a large proportion of the use of **PVM** will be constrained to local networks, with only a few applications wishing to execute components on geographically distant hosts. The protocols used by **PVM** therefore incur the overheads of retransmission and sequencing only when the underlying network quality is poor; more than 95% of local network communications typically succeed on the first attempt. Table 1 shows the message delivery times for varying message sizes under **PVM** between two Sun 3/50 systems on a 10 MB/s Ethernet. It should be noted that these figures represent elapsed time from the start of message transmission to the receipt of positive acknowledgment and are averages over several runs performed under varying host and network loads.

Message size (bytes)	8	128	256	512	1024
First instance	15 ms	18 ms	22 ms	25 ms	30 ms
Subsequent instances	6 ms	8 ms	10 ms	12 ms	15 ms

Table 1 : User process data transfer times in **PVM**

Broadcast, since it is used heavily within **PVM**, is another important factor in the performance of the system. It was observed that for broadcast among *pvmd* processes, the calculated performance of the recursive doubling algorithm is consistent with actual behavior. Acknowledged message transmission on a single branch of the broadcast spanning tree required between 4 and 9 milliseconds for a (typical) 100-byte message, depending upon the speed and load on the processors involved. This translated to measured figures of 15, 28, 35, and 50 milliseconds for typical broadcasts to 3, 7, 15, and 31 hosts respectively. For user process broadcasts, the figures vary widely, owing to the fact that *pvmd* processes deliver broadcasts sequentially to local recipients. Thus, if a large percentage of the user process broadcast group were physically executing on one host, the sequential delivery time for that host would dominate the total broadcast time. Table 2 shows typical time requirements for user broadcast, under the assumption that the broadcast groups are evenly distributed among participating hosts.

For access to shared memory segments, **PVM** performance was good, although the timings varied significantly, depending upon the number of active *pvmd* processes as well as the number of conflicting requests. It was empirically determined that the average time to access emulated shared memory could be expressed as

$$Time_{ms} = 12 \times \#\ of\ pvmd\ processes + 0.04 \times segment\_size_{bytes}$$

for shared memory segments of size 256 bytes and larger. the first term in the expression is that required for locking the segment, this term is dominant and increases with the number of distinct participating processors. An alternative mutual exclusion algorithm that reduced the effect of the

No. of processors	No. of processes	Message size (bytes) 8	128	256	512	1024
4	4	18 ms	22 ms	26 ms	30 ms	35 ms
4	8	21 ms	26 ms	30 ms	35 ms	41 ms
8	8	39 ms	48 ms	60 ms	75 ms	90 ms
8	16	47 ms	57 ms	70 ms	86 ms	105 ms
16	16	65 ms	76 ms	91 ms	110 ms	130 ms
16	32	95 ms	110 ms	125 ms	145 ms	180 ms

Table 2 : User process broadcast timings in **PVM**

first term by about 40% is available and may be used, but the failure resilience properties of this protocol are considered inadequate.

Owing to the manner in which process initiation and mutual exclusion are implemented, the time taken for these operations are almost identical to that for 100-byte broadcasts between *pvmd* processes. For barrier synchronization, it was once again observed that the sequential handling of component instances on a local processor could be dominant, if there were a large number of participants on any one machine. To measure the performance of the barrier **PVM** primitive was tested using a sample application that invoked the **barrier** construct twice consecutively, with the second invocation being timed. Shown in Table 3 are the observed measurements.

No. of processors	No. of processes	Time (msecs)
4	4	24
4	8	58
8	8	55
8	16	120
16	16	120
16	32	270
32	32	268
32	64	605

Table 3 : Barrier synchronization timings in **PVM**

4.2. Example Applications

In order to compare the relative performance of the **PVM** system to existing multiprocessors, two existing parallel codes were ported to run on the system. These two applications were ported without any changes to the algorithm itself, and were executed under **PVM** on a collection of homogeneous machines. The first application was numerical integration using the rectangle rule. The results from this experiment were uninterestingly predictable; on **PVM**, scaling in the number of processors or the number of rectangles resulted in a linear performance increase. Furthermore, the performance ratio between **PVM** and the iPSC/2 multiprocessor for this problem was constant — and consistent with the inherent processor speed differences. The second application is Cholesky matrix factorization [16] — an application that has a relatively high communication to computation ratio. Table 4 shows the elapsed times for this problem run on a network of Sun 3/50 machines for varying problem sizes. Shown in parentheses adjacent to each timing are the corresponding times for running the same program on an Intel iPSC/2 hypercube. In this experiment, no attempt was made to place more than one component process on a host; however, it should be noted that each participating host was also being used simultaneously for general purpose editing, compilation, and other normal workstation activities.

No. of processors	Problem size (Order of Matrix) 100		200		500		1000	
2	6	(2)	35	(30)	260	(245)	1950	(1923)
4	7	(2)	22	(16)	130	(120)	990	(970)
8	9	(1.5)	17	(9)	75	(64)	610	(490)
16	14	(1)	12	(6)	46	(34)	342	(255)

Table 4 : Times (in seconds) for Cholesky factorization

The anomaly apparent in the first column of Table 4 was traced down to the nature of the Cholesky algorithm — for a small matrix, the simultaneous broadcasts of every process's matrix column to a (relatively) large number of processors resulted in a high percentage of dropped packets, leading to retransmissions and elapsed timeouts. For larger problems however, it can be seen that the performance of **PVM** is acceptable at the least, considering that the application is a substantially communication oriented one and that general purpose machines on a local network were used. It should also be pointed out that these figures are 2 to 4 times better than those for other distributed multiprocessor simulators such as dsim[5]. Furthermore, the factorization program was built for performance measurement purposes and therefore internally generated the matrix elements and did not output the factorized results. Given the usual difficulty and inefficiencies in I/O from within the nodes in a distributed memory machine, it is expected that **PVM** will compare much more favorably against hardware multiprocessors when significant amounts of I/O are performed.

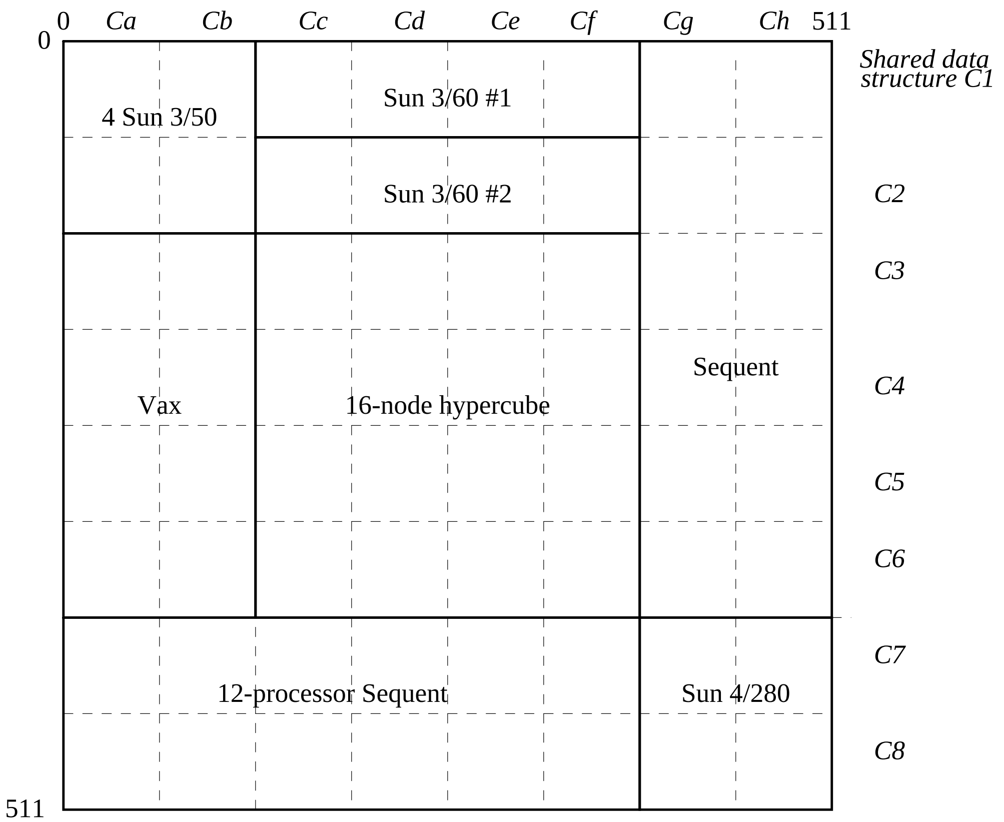
5. Application Experiences

The **PVM** system has evolved and has been in intermittent use for the past several months, and enhancements are still ongoing. Several application codes have been ported to run under the **PVM** environment; in this section, two applications that make extensive use of the system features are described.

5.1. Block Matrix Multiplication

Matrix multiplication is a compute intensive application that is amenable to parallel solution. On shared memory multiprocessors, prescheduling algorithms work well and exhibit good speedup characteristics with an increase in the number of processors. On distributed memory machines, matrices are decomposed into subblocks and multiplied; and a regular communication pattern between the processing elements helps minimize the overheads. A detailed description and analysis of block matrix multiplication on hypercube architectures may be found in [18].

A modified version of the block matrix multiplication algorithm was implemented on the **PVM** system and executed on various hardware combinations. The modification was with regard to subblock decomposition; when the individual processors differ widely in computing capabilities, equal sized subblocks lead to severe load imbalances and result in significant performance degradation. To overcome this, the matrices to be multiplied were viewed as being composed of small, equal sized subblocks — with individual processors taking responsibility for *collections* of subblocks at a time, the size of the collections being determined by their relative computing power. In other words, the algorithm was expressed using the "bag of tasks" approach, where each task is individual subblock multiplication. However, each worker process extracts, or assumes responsibility for, a number of tasks at a time to reduce synchronization and cooperation overheads.



- Dashed grid shows decomposition
-Heavy boxes show actual allocation for result matrix subblock computations

-Each Cn is a 64x512 "band"
- Bands in A and B are similarly identified

Figure 7: Asymteric decomposition for matrix multiplication

Figure 7 shows an example scenario, where task partitioning is shown in terms of the resulting matrix. It should be noted that the decomposition shown in the figure is not statically determined; each processor computes one subblock aggregate, then obtains another, and so on. To implement this on **PVM**, the following strategy was used:

- The matrices to be multiplied (*A* and *B*) are maintained in emulated shared memory by the "startup" process, with each subblock row and column (band) being independently identifiable as shown in Figure 7. The granularity of sharing is therefore large.
- System component programs are written to use the algorithm most appropriate for their targeted architecture, i.e. a square subblock component for the hypercube, a static prescheduled component for the Sequent shared memory multiprocessor, and sequential code for the Sun workstations.
- A "working list" shared data structure is set up, once again by the "startup" process. This list holds the result matrix subblocks that have been computed or have been taken up by a component. For example, when all but the bottom right corner subblock of *C* (Fig. 7) has been computed the list might contain:
    + {0,127},{0,127} : Indicating that the 4-member group of Sun 3/50's (using **PVM** primitives to cooperate using message-passing) had taken responsibility for the top left corner subblock.
    + {128,383},{0,63} : Taken (and perhaps completed) by Sun 3/60 #1.
    + {128,383},{65,127} : Taken by Sun 3/60 #2.
    + {384,511},{0,383} : Taken by and completed by the Sequent.
    + {0,127},{128,383} : Taken by Vax.
    + {0,383},{384,511} : Taken by Sequent and perhaps being currently processed.

  The last processor, upon inspecting this list, determines that it can compute the remaining subblock ({384,511},{384,511}), and enters this pair in the list - thereby signaling completion to other processors.
- Having determined which subblock a component was going to compute, the component then maps the appropriate segments of the *A* and *B* matrices into its local memory. Again using figure 7 as an example, the component executing on "Sun 3/60 #2" would map the bands A2 and Bb-Bf into its local memory, and using these, would compute the shown subblock of the matrix *C*. The Sequent (in one case) would map bands A7-A8 and Ba-Bf, and use a locally parallel program to compute the resulting matrix subblock.

The results of executing this asymmetric algorithm using multiple architectures are shown in Table 5. Also shown are the timing figures for multiplying the same matrices using "pure" algorithms on a hypercube and a shared memory machine. It can be seen from the figure that the **PVM** implementation scales well, even for a heterogeneous collection of processors, when the matrices are large. For smaller matrices however, the locking and emulated shared memory overheads actually cause an *increase* in execution times when more processors are added. Nevertheless, it is felt that such algorithms are valuable for two important reasons. First, when multiple processors are available, the total computing power can be increased — e.g. a 16-node hypercube may be combined with other processors to emulate a 32 processor virtual machine that is beneficial for large matrices. Second, the the support services that are available on the network

Machine type	No. of procs.	Problem Size (Order of Matrix) 64	128	256	512
Sequent	4	5.4	40.5	326.6	2581.9
Sequent	8	2.9	20.9	163.8	1292.7
iPSC/2	4	1.5	9.3	81.5	693.5
iPSC/2	16	1.2	3.1	18.2	191.7
PVM (Suns)	4	1.8	4.3	26.5	231.6
PVM (Suns+ Sequent-8)	16	2.8	5.1	20.3	212.5
PVM(Suns+ Sequent-8+ iPSC/2-16)	32	3.5	7.2	16.6	130.8

Table 5: Times (in seconds) for matrix multiplication

(such as filesystems etc.) may be exploited; for very large datasets, asymmetric partitioning algorithms can contribute significantly to reduced I/O and swapping overheads.

5.2. Contaminant Transport Simulation

To further exercise the heterogeneous computing facilities in **PVM**, an interactive simulation program was ported to the system. This simulation models contaminant transport in groundwater flow, and is part of a larger and more comprehensive simulation effort. Preliminary findings from this work are reported in [2]. The major components of the existing software are a graphical interface and the solution of sets of differential equations involving transport calculations and flow calculations. The transport calculations are highly parallelizable, while the flow equations may be well solved on a fast scalar machine. The existing implementation encountered several difficulties, including the interfacing of the two computational components, the transfer of structured data between multiple instances, and interactions with an X-windows graphical interface. The latter issue was particularly cumbersome when the system was run over a wide area network.

The simulation programs were ported for **PVM** use, by partitioning the system into three components - the transport calculations (executed on a hypercube), the flow equations (on a fast scalar machine), and the graphical interface on a Sun workstation. Very few modifications in the existing codes were required; the scalar and hypercube components interacted via emulated shared memory while the graphics component used the fast message passing primitives in **PVM** to achieve better response and lowered latency. An additional benefit was the mobility of the graphical interface component; due to the connectionless nature of **PVM** interactions, it was possible to execute the graphical interface program from different locations in a manner transparent to the computational components. Further, multiple displays were possible by using the **PVM** broadcast facilities, a useful feature when several simultaneous users wish to interact with the system. Due to the interactive nature of the application, performance timings are not meaningful; however, the system performed without any noticeable degradation when executed under **PVM**.

6. Conclusions and Future Work

The primary motivation for the **PVM** project is derived from the existing and anticipated need for a general, flexible, and inexpensive concurrent programming environment. Experiences with the system have demonstrated that such a framework can be provided and can execute on existing hardware bases, with the benefits of a procedural programming interface and straightforward constructs for access to various resources. A noteworthy feature of **PVM** is its generality — supporting both the shared-memory and message-passing paradigms on a heterogeneous collection of machines is useful. The framework provided by **PVM** enables interaction between application components and machine architectures that are normally incompatible. Anticipating that large and complex parallel systems will require error indication and failure detection capabilities, such features have been built into the **PVM** system and are likely to be valuable. From the performance point of view, **PVM** has proven to be acceptable even for applications with a high communication to computation ratio — although its primary intent is to support applications with much larger grainsize and less interaction. Perhaps of more importance in certain situations is the ability of **PVM** to utilize resources that already exist and would be wasted otherwise, not to mention its value as a prototyping tool for new algorithms or applications. The simplicity of porting the **PVM** system as well as application software also enhances its appeal and will contribute to its increased use.

There are, however, several aspects to **PVM** that require further work; some efforts are ongoing while others are planned for the future. A very valuable addition would be a supplemental toolkit that assisted in algorithm partitioning and scheduling, as well as support for the automatic compilation of different object modules for different architectures. From the system implementation point of view, it is evident that the data transfer, broadcast, and mutual exclusion protocols are the most crucial primitives, and work is in progress to optimize these. Conflict resolution in the locking algorithm presently uses a heuristic method — a provably correct formalization of this will be undertaken soon. From the application point of view, certain additional features might be desirable such as (1) the ability to coalesce emulated and real shared memory, and (2) to dynamically optimize message passing, locking etc., depending on the architecture on which a component instance is executing. Also planned for the future are a graphical interface for the specification of component execution order and interactions, as well as debugging and execution history trace facilities.

Acknowledgments

The author is indebted to the referees, particularly the second referee, for detailed comments that resulted in many improvements in the system and this paper. Thanks are also due to M. T. Heath, G. A. Geist, T. H. Dunigan, and D. A. Poplawski for helpful discussions during the course of this work and for their comments on earlier versions of this paper.

References

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

[2] H. Narang, R. Flanery, J. Drake, *Design of a Simulation Interface for a Parallel Computing Environment*, Proc. ACM Southeastern Regional Conference, April 1990, to appear.

[3] G. A. Geist, M. T. Heath, B. W. Peyton, P. H. Worley, *A Machine Independent Communication Library*, Proc. Hypercube Concurrent Computers Conference 1989, J. Gustafson ed., to appear.

[4] T. J. Gardner, *et. al.*, *DPUP : A Distributed Processing Utilities Package*, Computer Science Technical Report, University of Colorado, Boulder, 1986.

[5] T. H. Dunigan, *Hypercube Simulation on a Local Area Network*, Oak Ridge National Laboratory Report ORNL/TM-10685, November 1988.

[6] C. Seitz, J. Seizovic, W. K. Su, *The C programmer's Abbreviated Guide to Multicomputer Programming*, Caltech Computer Science Report, CS-TR-88-1, January 1988.

[7] M. Sullivan, D. Anderson, *Marionette: A System for Parallel Distributed Programming Using a Master/Slave Model*, Proc. 9th ICDCS, June 1989, pp. 181-188.

[8] G. Riccardi, B. Traversat, U. Chandra, *A Master-Slaves Parallel Computation Model*, Supercomputer Research Institute Report, Florida State University, June 1989.

[9] Intel iPSC/2 Programmer's Reference Manual, Intel Corporation, Beaverton, March 1988.

[10] A. Karp, *Programming for Parallelism*, IEEE Computer, May 1987, pp. 43-57.

[11] J. B. Postel, *User Datagram Protocol*, Internet Request for Comments RFC-768, August 1980.

[12] N. Maekawa, *A $\sqrt{n}$ Algorithm for Mutual Exclusion in Decentralized Systems, ACM Transactions on Computer Systems, May 1985, pp. 145-159.*

[13] K. Raymond, *A Tree Based Algorithm for Distributed Mutual Exclusion*, ACM Transactions on Computer Systems, February 1989, pp. 61-77.

[14] D. Agarwal, A. E. Abbadi, *An Efficient Solution to the Distributed Mutual Exclusion Problem*, Proc. Principles of Distributed Computing Conference, Edmonton, August 1989.

[15] J. S. Chase *et. al.*, *The Amber System: Parallel Programming on a Network of Multiprocessors*, to appear in 12th SOSP, Litchfield Park, November 1989.

[16] G. A. Geist, M. T. Heath, *Matrix Factorization on Hypercube Multiprocessors*, in Hypercube Multiprocessors 1986, SIAM, Philadelphia, 1986, pp. 161-180.

[17] D. W. Lozier, R. G. Rehm, *Some Performance Comparisons for a Fluid Dynamics Code*, Parallel Computing, Vol. 11, pp. 305-320, 1989.

[18] G. C. Fox, S. W. Otto, *Matrix Algorithms on a Hypercube I: Matrix Multiplication*, Parallel Computing, Vol. 4, pp. 17-31, 1987.

[19] A. Osterhaug, *Guide to Parallel Programming on Sequent Computer Systems*, Sequent Computer Systems, Inc., Beaverton, 1986.

[20] R. Finkel, E. Styer, U. Manber, *Designing Efficient Barriers in Communication Networks*, University of Kentucky Technical Report, No. 165-90, March 1990.

Appendix — PVM User Interface Constructs

enroll(<*name*>)

Enroll calling process as <*name*>. Returns instance number.

initiate(<*component name*>)

Start new process specified by <*component name*>. Returns instance number.

initiateP(<*name1*>,<*name2*>,<*inum*>)

Start new instance of <*name1*> when <*name2*>/<*inum*> terminates.

initiateD(<*name*>,<*event*>)

Start new instance of <*name*> when <*event*> occurs.

ready(<*event*>)

Inform **PVM** system of occurrence of <*event*>.

terminate(<*name*>,<*inum*>)

Terminate instance <*inum*> of component <*name*>.

waitprocess(<*name*>,<*inum*>)

Block caller until instance <*inum*> of <*name*> terminates.

send(<*name*>,<*inum*>,<*type*>)

Send message of specified type to specified destination process. Negative return value on failure.

recv(<*type*>)

Receive message of specified type.

recv1(<*type*>,<*other_limit*>)

Receive message of specified type; <*other_limit*> msgs of other types allowed. Negative return value on failure.

recv2(<*type1*>,<*timeout*>)

Receive message of specified type within <*timeout*> seconds. Negative return value on failure.

putstring(<*string*>)

Store <*string*> in send buffer in machine independent form.

putint(<*num*>)

Store integer in send buffer in machine independent form.

putfloat(<*fnum*>)

Store real number in send buffer in machine independent form.

getstring(<*string_ptr*>)

Retrieve string from receive buffer in machine dependent form.

getint(<*integer_ptr*>)

Retrieve integer from receive buffer in machine dependent form.

getfloat(<*float_ptr*>)

Retrieve real number from receive buffer in machine dependent form.

recvinfo(<*string_ptr*>,<*inum_ptr*>,<*len_ptr*>)

Return source name, instance number, and length of last received message.

broadcast(<*name*>)

Broadcast send buffer to all instances of <*name*>. Returns actual number of recipients.

barrier(<*number*>)

Blocks caller until specified number of instances arrive at barrier. Negative return value if some instances have terminated.

waituntil(<*event*>)

Blocks caller until specified event occurs.

shmget(<*key*>,<*size*>)

Allocates shared memory segment of specified size identified by <*key*>. A negative return value indicates that the key value is already in use.

shmat(<*key*>,<*ptr*>,<*flag*>,<*timeout*>)

Locks shared memory segment identified by <*key*> and maps segment at caller's address space starting at *ptr*>. "R" or "RW" are possible flag values; a negative value is returned if the attach does not succeed within <*timeout*> seconds. A 0 timeout value causes the caller to block until successful.

shmatint(<*key*>,<*integer_ptr*>,<*flag*>,<*timeout*>)

Variant of **shmat** that maps segment in typed form as integer array.

shmatfloat(<*key*>,<*float_ptr*>,<*flag*>,<*timeout*>)

Variant of **shmat** that maps segment in typed form as float array.

shmdt(<*key*>,<*ptr*>)

Unlocks and unmaps specified shared memory segment from process' address space indicated by <*ptr*>.

shmdtint(<*key*>,<*integer_ptr*>)

Unlocks and unmaps specified shared memory segment in a typed form from process' address space indicated by <*integer_ptr*>.

shmdtfloat(<*key*>,<*float_ptr*>)

Unlocks and unmaps specified shared memory segment in a typed form from process' address space indicated by <*float_ptr*>.

shmfree(<*key*>)

Deallocates specified shared memory segment.

lock(<*resource_name*>,<*timeout*>)

Permits exclusive access to abstract resource identified by string-valued <*resource_name*>. Negative return value indicates resource could not be acquired within specified timeout period.

unlock(<*resource_name*>)

Releases lock on previously acquired resource.

status(<*component_name*>,<*inum*>, <*stat_ptr*>,<*loc_ptr*>)

Takes string valued component name and an instance number and returns the status (0=nonexistent, 1=active) and location (processor number in the range 0 — p) of that component instance.

entercomp(<*name*>,<*loc_machine*>,<*obj_file*>,<*arch*>)

Permits a component description to be added; specifying the component name, the object file name and the machine on which it is located, and the type of architecture that the object will execute on.

shmstat(<*key_ptr_ptr*>,<*stat_ptr*>)

Obtains information about shared memory regions. Array of strings holds the key values; array of integers holds status (0=free, 1=locked) information. Return value specifies total number of active regions.

lockstat(<*key_ptr_ptr*>)

Returns total number of active locks with array of strings holding the key values.

Notes:

Among the **PVM** constructs described above, those concerned with machine dependent data representation are to be implemented as part of the installation procedure for architectures not represented in the generic distribution of the software. Also deemed part of this procedure are constructs to handle other data types such as double precision, boolean, enumerated types, etc.