NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

PARALLEL VIRTUAL MACHINE SYSTEM

Graduation Project
Com – 400

Student : Serdar AYDIN

Supervisor : Mr. Halil ADAHAN

Nicosia - 2003

ACKNOWLEDGE

I would like to thank all those who offered encouragement and advice throughout this project. Expecially, I'd like to thank Halil ADAHAN for advised and helped. Also I would like to thank Arzu YILMAZ sterling support and conversation.In particular, my family offered helpful comments along the way. Last, but no means least,I would like to thank Muhsin GEMICI for helping. If there were no in my side,it would be hard and take a long time.

CONTENTS

CONTENT OF THE FIGURE

CONTENT OF THE TABLE

ABSTRACT

The PVM (Parallel Virtual Machine) system is a programming environment for the development and execution of large concurrent or parallel application that consists of many interacting, but relatively independent, components. It is also named as, a software framework for heterogeneous concurrent computing in networked environments has evolved into a viable technology for distributed and parallel using the combined resources of heterogeneous networked computing platforms to deliver high multiprocessors, or general-purpose computers, enabling application components to execute on of the PVM system, its computing model, programming interface it supports, and typical application implementation such as matrix multiplication using MPP.

INTRODUCTION

Parallel processing, the method of having many small tasks solve one large problem, has emerged as a key enabling technology in modern computing. The past several years have witnessed an ever-increasing acceptance and adoption of parallel processing, both for high-performance scientific computing and for more "general-purpose" applications, was a result of the demand for higher performance, lower cost, and sustained productivity. The acceptance has been facilitated by two major developments: massively parallel processors (MPPs) and the widespread use of distributed computing.

MPPs are now the most powerful computers in the world. These machines combine a few hundred to a few thousand CPUs in a single large cabinet connected to hundreds of gigabytes of memory. MPPs offer enormous computational power and are used to solve computational Grand Challenge problems such as global climate modeling and drug design. As simulations become more realistic, the computational power required to produce them grows rapidly. Thus, researchers on the cutting edge turn to MPPs and parallel processing in order to get the most computational power possible.

The second major development affecting scientific problem solving is *distributed computing* . Distributed computing is a process whereby a set of computers connected by a network are used collectively to solve a single large problem. As more and more

organizations have high-speed local area networks interconnecting many general-purpose workstations, the combined computational resources may exceed the power of a single high-performance computer. In some cases, several MPPs have been combined using distributed computing to produce unequaled computational power.

The most important factor in distributed computing is cost. Large MPPs typically cost more than $10 million. In contrast, users see very little cost in running their problems on a local set of existing computers. It is uncommon for distributed-computing users to realize the raw computational power of a large MPP, but they are able to solve problems several times larger than they could using one of their local computers.

Common between distributed computing and MPP is the notion of message passing . In all parallel processing, data must be exchanged between cooperating tasks. Several paradigms have been tried including shared memory, parallelizing compilers, and message passing. The message-passing model has become the paradigm of choice, from the perspective of the number and variety of multiprocessors that support it, as well as in terms of applications, languages, and software systems that use it.

The Parallel Virtual Machine (PVM) system described in this book uses the message-passing model to allow programmers to exploit distributed computing across a wide variety of computer types, including MPPs. A key concept in PVM is that it makes a collection of computers appear as one large *virtual* machine , hence its name.

The PVM system is a software infrastructure that evolves a generalized distributed memory multiprocessor in heterogeneous networked environments. The PVM system has gained acceptance and adoption of parallel processing, both for high scientific computing as well as for more "general-purpose" applications. Moreover, the message-passing model of PVM has gained predominance from the perspective of the number of variety of multiprocessors, and also in term of application, languages, and software system. The PVM approach has proven to be a viable and cost-effective technology for concurrent computing in many application domains. The PVM system has gained widespread

acceptance, to the extent that is referred to as the "*de-facto*" standard for distributed memory concurrent computing.

Many application that are adaptable to concurrent execution can be programmed using either message passing or share memory algorithms. A hybrid algorithm, at a sufficiently high level of granularity may be used when computing supporting environments are available. As an example is matrix multiplication. 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 rescheduling or shared memory machines strategies on distributed memory.

Most typical computing environments already process the hardware diversity to such large, parallel applications, and also contain support for multiple concurrent computation models. High speed local networks and graphics workstation, high-performance scalar engines, and vector computer are that norm rather than the exception, and will continue to be over the coming years. However, to implement this collection of capabilities and to utilize it productively requires considerable efforts is coordination and reconciliation between different computation models and architectures. The PVM system attempts to provide a unified framework within which a objective of the PVM system is to permit a collection of heterogeneous machines on a network to be viewed as a general-purpose concurrent computation resource.

The PVM system provides a set of user interface primitives that may be incorporated into existing 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 conditions upon the initiation or termination of another process, or upon the availability of data values. The PVM constructs 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.

1.2 Heterogeneous Network Computing

Heterogeneous, network-base, concurrent computing refers to an evolving methodology for general-purpose concurrent computing, described by two major actions.

1. Interconnected by one or more network types, the hardware platform consists of a collection of multifaceted computer system of varying architectures.
2. Applications are viewed as comprising several sub-algorithms, each of, which is potentially different in terms of its most appropriate programming model, implementation language, and resource requirements

Heterogeneous network computing refers to models, techniques, and toolkits to match heterogeneous environments with complete applications, consisting of different subtask. The PVM system was designed to realize a more general and encompassing interpretation of heterogeneous computing. PVM supports heterogeneous machines, applications and networks. The future extension of PVM research is mainly to propose a heterogeneous application development model and associated programming frameworks, and to provide adequate infrastructure for heterogeneous debugging, utilization, profiling and monitoring. In high performance of scientific and robust emulation's of heterogeneous concurrent machines, have proven to be valuable and effective for more traditional application.

CHAPTER 1: The PVM SYSTEM

PVM (Parallel Virtual Machine) is a byproduct of an ongoing heterogeneous network computing research project involving the authors and their institutions. The general goals of this project are to investigate issues in, and develop solutions for, heterogeneous concurrent computing. PVM is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architecture. The overall objective of the PVM system is to enable such a collection of computers to be used cooperatively for concurrent or parallel computation. Detailed descriptions and discussions of the concepts, logistics, and methodologies involved in this network-based computing process are contained in the remainder of the book. Briefly, the principles upon which PVM is based include the following:

1. User-configured host pool : The application's computational tasks execute on a set of machines that are selected by the user for a given run of the PVM program. Both single-CPU machines and hardware multiprocessors (including shared-memory and distributed-memory computers) may be part of the host pool. The host pool may be altered by adding and deleting machines during operation (an important feature for fault tolerance).
2. Translucent access to hardware: Application programs either may view the hardware environment as an attributes collection of virtual processing elements or may choose to exploit the capabilities of specific machines in the host pool by positioning certain computational tasks on the most appropriate computers.
3. Process-based computation: The unit of parallelism in PVM is a task (often but not always a Unix process), an independent sequential thread of control that alternates between communication and computation. No process-to-processor mapping is implied or enforced by PVM; in particular, multiple tasks may execute on a single processor.
4. Explicit message-passing model: Collections of computational tasks, each performing a part of an application's workload using data-, functional-, or hybrid decomposition,

cooperate by explicitly sending and receiving messages to one another. Message size is limited only by the amount of available memory.

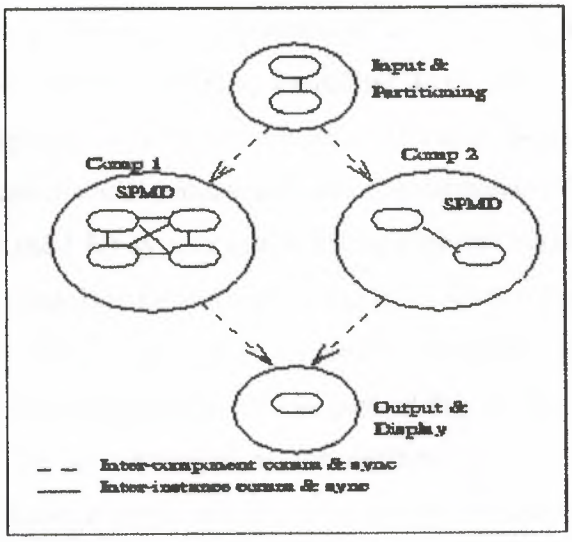
5. Heterogeneity support: The PVM system supports heterogeneity in terms of machines, networks, and applications. With regard to message passing, PVM permits messages containing more than one data type to be exchanged between machines having different data representations.
6. Multiprocessor support: PVM uses the native message-passing facilities on multiprocessors to take advantage of the underlying hardware. Vendors often supply their own optimized PVM for their systems, which can still communicate with the public PVM version.

Key Featues

1. Easily obtainable public domain package.
2. Easy to install.
3. Easy to configure.
4. Many different virtual machines may co- exist on the same hardware.
5. Program development using a widely adopted message passing library.
6. Supports C and Fortran.
7. Installation only requires a few Mb of disk space.
8. Simple migration path to MPI.

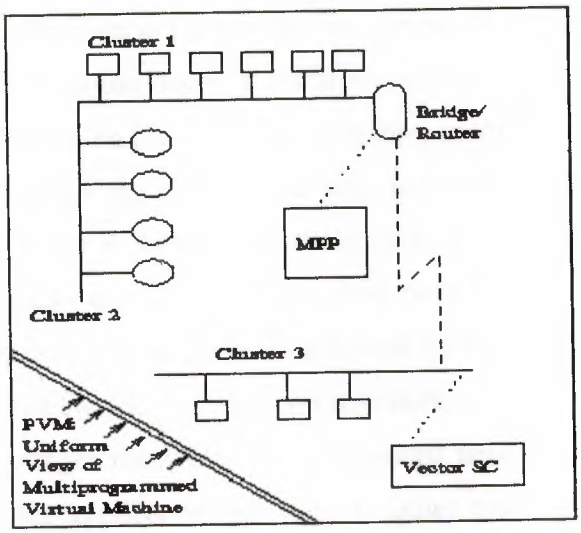
The PVM system is composed of two parts. The first part is a daemon , called *pvmd3* and sometimes abbreviated *pvmd* , that resides on all the computers making up the virtual machine. (An example of a daemon program is the mail program that runs in the background and handles all the incoming and outgoing electronic mail on a computer.) Pvmd3 is designed so any user with a valid login can install this daemon on a machine. When a user wishes to run a PVM application, he first creates a virtual machine by starting up PVM. The PVM application can then be started from a Unix prompt on any of the hosts. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously. The second part of the system is a library of PVM interface routines. It contains a functionally complete repertoire of primitives that are needed for cooperation between tasks of an application. This library

contains user-callable routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine. The PVM computing model is based on the notion that an application consists of several tasks. Each task is responsible for a part of the application's computational workload. Sometimes an application is parallelized along its functions; that is, each task performs a different function, for example, input, problem setup, solution, output, and display. This process is often called functional parallelism . A more common method of parallelizing an application is called data parallelism . In this method all the tasks are the same, but each one only knows and solves a small part of the data. This is also referred to as the SPMD (single-program multiple-data) model of computing. PVM supports either or a mixture of these methods. Depending on their functions, tasks may execute in parallel and may need to synchronize or exchange data, although this is not always the case. An exemplary diagram of the PVM computing model is shown in Figure1 and an architectural view of the PVM system, highlighting the heterogeneity of the computing platforms supported by PVM, is shown in Figure 1.1



(a) PVM Computation Model

Figure 1.1: PVM System Overview



(b) PVM Architectural Overview

Figure 1.2: PVM System Overview

The PVM system currently supports C, C++, and Fortran languages. This set of language interfaces have been included based on the observation that the predominant majority of target applications are written in C and Fortran, with an emerging trend in experimenting with object-based languages and methodologies. The C and C++ language bindings for the PVM user interface library are implemented as functions, following the general conventions used by most C systems, including Unix-like operating systems. To elaborate, function arguments are a combination of value parameters and pointers as appropriate, and function result values indicate the outcome of the call. In addition, macro definitions are used for system constants, and global variables such as errno and *pvm_errno* are the mechanism for discriminating between multiple possible outcomes. Application programs written in C and C++ access PVM library functions by linking against an archival library (libpvm3.a) that is part of the standard distribution. Fortran language bindings are implemented as subroutines rather than as functions. This approach was taken because some compilers on the supported architectures would not reliably interface Fortran functions with C functions. One immediate implication of this is that an additional argument is introduced into each PVM library call for status results to be returned to the invoking program. Also, library routines for the placement and retrieval of typed data in message buffers are unified, with an additional parameter indicating the data type. Apart from these differences (and the standard naming prefixes -

*pvm_* for C, and *pvmf* for Fortran), a one-to-one correspondence exists between the two language bindings. Fortran interfaces to PVM are implemented as library stubs that in turn invoke the corresponding C routines, after casting and/or dereferencing arguments as appropriate. Thus, Fortran applications are required to link against the stubs library (libfpvm3.a) as well as the C library. All PVM tasks are identified by an integer *task identifier* (TID). Messages are sent to and received from TID. Since TID must be unique across the entire virtual machine, they are supplied by the local *pvmd* and are not user chosen. Although PVM encodes information into each TID the user is expected to treat the TID as opaque integer identifiers. PVM contains several routines that return TID values so that the user application can identify other tasks in the system. There are applications where it is natural to think of a *group of tasks* . And there are cases where a user would like to identify his tasks by the numbers **0 - (p - 1)**, where **p** is the number of tasks. PVM includes the concept of user named groups. When a task joins a group, it is assigned a unique "instance" number in that group. Instance numbers start at 0 and count up. In keeping with the PVM philosophy, the group functions are designed to be very general and transparent to the user. For example, any PVM task can join or leave any group at any time without having to inform any other task in the affected groups. Also, groups can overlap, and tasks can broadcast messages to groups of which they are not a member. To use any of the group functions, a program must be linked with *libgpvm3.a* .

The general paradigm for application programming with PVM is as follows. A user writes one or more sequential programs in C, C++, or Fortran 77 that contain embedded calls to the PVM library. Each program corresponds to a task making up the application. These programs are compiled for each architecture in the host pool, and the resulting object files are placed at a location accessible from machines in the host pool. To execute an application, a user typically starts one copy of one task (usually the "master" or "initiating" task) by hand from a machine within the host pool. This process subsequently starts other PVM tasks, eventually resulting in a collection of active tasks that then compute locally and exchange messages with each other to solve the problem. Note that while the above is a typical scenario, as many tasks as appropriate may be started

manually. As mentioned earlier, tasks interact through explicit message passing, identifying each other with a system-assigned, opaque TID.

```
main()

{
  int cc, tid, msgtag;
  char buf[100];

  printf("i'm t%x\n", pvm_mytid());

  cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

  if (cc == 1) {
     msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");

  pvm_exit();
}
```

This program is intended to be invoked manually; after printing its task id (obtained with *pvm_mytid()*), it initiates a copy of another program called *hello_other* using the *pvm_spawn()* function. A successful spawn causes the program to execute a blocking receive using *pvm_recv*. After receiving the message, the program prints the message sent by its counterpart, as well its task id; the buffer is extracted from the message using *pvm_upkstr*. The final *pvm_exit* call dissociates the program from the PVM system.

```
#include "pvm3.h"
```

```
main()
{
  int ptid, msgtag;
  char buf[100];

  ptid = pvm_parent();

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
   msgtag = 1;
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

Its first PVM action is to obtain the task id of the "master" using the *pvm_parent* call. This program then obtains its hostname and transmits it to the master using the three-call sequence - *pvm_initsend* to initialize the send buffer; *pvm_pkstr* to place a string, in a strongly typed and architecture-independent manner, into the send buffer; and *pvm_send* to transmit it to the destination process specified by *ptid,* "tagging" the message with the number 1.

1.1. PVM User Interface

In this chapter we give a brief description of the routines in the PVM 3 user library. This chapter is organized by the functions of the routines. For example, *Message Passing* is a discussion of all the routines for sending and receiving data from one PVM task to another and a description of PVM's message passing options. The calling syntax of the C and Fortran PVM routines are highlighted by boxes in each section.

In PVM 3 all PVM tasks are identified by an integer supplied by the local *pvmd*. In the following descriptions this task identifier is called TID. It is similar to the process ID (PID) used in the Unix system and is assumed to be opaque to the user, in that the value of the TID has no special significance to him. In fact, PVM encodes information into the TID for its own internal use

All the PVM routines are written in C. C++ applications can link to the PVM library. Fortran applications can call these routines through a Fortran 77 interface supplied with the PVM 3 source. This interface translates arguments, which are passed by reference in Fortran, to their values if needed by the underlying C routines. The interface also takes into account Fortran character string representations and the various naming conventions that different Fortran compilers use to call C functions.

The PVM communication model assumes that any task can send a message to any other PVM task and that there is no limit to the size or number of such messages. While all hosts have physical memory limitations that limits potential buffer space, the communication model does not restrict itself to a particular machine's limitations and assumes sufficient memory is available. The PVM communication model provides asynchronous blocking send, asynchronous blocking receive, and non-blocking receive functions. In our terminology, a blocking send returns as soon as the send buffer is free for reuse, and an asynchronous send does not depend on the receiver calling a matching receive before the send can return. There are options in PVM 3 that request that data be transferred directly from task to task. In this case, if the message is large, the sender may block until the receiver has called a matching receive.

A non-blocking receive immediately returns with either the data or a flag that the data has not arrived, while a blocking receive returns only when the data is in the receive buffer. In addition to these point-to-point communication functions, the model supports multicast to a set of tasks and broadcast to a user-defined group of tasks. There are also functions to perform global max, global sum, etc., across a user-defined group of tasks. Wildcards can be specified in the receive for the source and label, allowing either or both of these

contexts to be ignored. A routine can be called to return information about received messages.

The PVM model guarantees that message order is preserved. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both messages arrive before task 2 does a receive, then a wildcard receive will always return message A.

Message buffers are allocated dynamically. Therefore, the maximum message size that can be sent or received is limited only by the amount of available memory on a given host. There is only limited flow control built into PVM 3.3. PVM may give the user a *can't get memory* error when the sum of incoming messages exceeds the available memory, but PVM does not tell other tasks to stop sending to this host.

1.1.1 Message Passing

Sending a message comprises three steps in PVM. First, a send buffer must be initialized by a call to *pvm_initsend()* or *pvm_mkbuf()*. Second, the message must be "packed" into this buffer using any number and combination of *pvm_pk*()* routines. (In Fortran all message packing is done with the *pvmfpack()* subroutine.) Third, the completed message is sent to another process by calling the *pvm_send()* routine or multicast with the *pvm_mcast()* routine. A message is received by calling either a blocking or non-blocking receive routine and then "unpacking" each of the packed items from the receive buffer. The receive routines can be set to accept *any* message, or any message from a specified source, or any message with a specified message tag, or only messages with a given message tag from a given source. There is also a probe function that returns whether a message has arrived, but does not actually receive it.

If required, other receive contexts can be handled by PVM 3. The routine *pvm_recvf()* allows users to define their own receive contexts that will be used by the subsequent PVM receive routines.

1.2 How PVM Works

This section describes the implementation of the PVM software and the reasons behind the basic design decisions. The most important goals for PVM 3 are fault tolerance, scalability, heterogeneity, and portability. PVM is able to withstand host and network failures. It doesn't automatically recover an application after a crash, but it does provide polling and notification primitives to allow fault-tolerant applications to be built. The virtual machine is dynamically reconfigurable. This property goes hand in hand with fault tolerance: an application may need to acquire more resources in order to continue running once a host has failed. Management is as decentralized and localized as possible, so virtual machines should be able to scale to hundreds of hosts and run thousands of tasks. PVM can connect computers of different types in a single session. It runs with minimal modification on any flavor of Unix or an operating system with comparable facilities (multitasking, networkable). The programming interface is simple but complete, and any user can install the package without special privileges.

To allow PVM to be highly portable, we avoid the use of operating system and language features that would be hard to retrofit if unavailable, such as multithreaded processes and asynchronous I/O. These exist in many versions of Unix, but they vary enough from product to product that different versions of PVM might need to be maintained. The generic port is kept as simple as possible, though PVM can always be optimized for any particular machine.

We assume that *sockets* are used for inter-process communication and that each host in a virtual machine group can connect directly to every other host via TCP [9] and UDP [10] protocols. The requirement of full IP connectivity could be removed by specifying message routes and using the pvmds to forward messages. Some multiprocessor machines don't make sockets available on the processing nodes, but do have them on the front-end .

1.3 Setup to Use PVM

One of the reasons for PVM's popularity is that it is simple to set up and use. PVM does not require special privileges to be installed. Anyone with a valid login on the hosts can do so. In addition, only one person at an organization needs to get and install PVM for everyone at that organization to use it.

PVM uses two environment variables when starting and running. Each PVM user needs to set these two variables to use PVM. The first variable is PVM_ROOT , which is set to the location of the installed pvm3 directory. The second variable is PVM_ARCH , which tells PVM the architecture of this host and thus what executables to pick from the PVM_ROOT directory.

The easiest method is to set these two variables in your .cshrc file. We assume you are using *csh* as you follow along this tutorial. Here is an example for setting PVM_ROOT:

setenv PVM_ROOT $HOME/pvm3

It is recommended that the user set PVM_ARCH by concatenating to the file .cshrc, the content of file $PVM_ROOT/lib/cshrc.stub. The stub should be placed after PATH and PVM_ROOT are defined. This stub automatically determines the PVM_ARCH for this host and is particularly useful when the user shares a common file system (such as NFS) across several different architectures.

PVM_ARCH	Machine	Notes
AFX8	Alliant FX/8	
ALPHA	DEC Alpha	DEC OSF-1
BAL	Sequent Balance	DYNIX
BFLY	BBN Butterfly TC2000	
BSD386	80386/486 PC runnning Unix	BSDI, 386BSD, NetBSD
CM2	Thinking Machines CM2	Sun front-end
CM5	Thinking Machines CM5	Uses native messages

CNVX	Convex C-series	IEEE f.p.
CNVXN	Convex C-series	native f.p.
CRAY	C-90, YMP, T3D port available	UNICOS
CRAY2	Cray-2	
CRAYSMP	Cray S-MP	
DGAV	Data General Aviion	
E88K	Encore 88000	
HP300	HP-9000 model 300	HPUX
HPPA	HP-9000 PA-RISC	
I860	Intel iPSC/860	Uses native messages
IPSC2	Intel iPSC/2 386 host	SysV, Uses native messages
KSR1	Kendall Square KSR-1	OSF-1, uses shared memory
LINUX	80386/486 PC running Unix	LINUX
MASPAR	Maspar	
MIPS	MIPS 4680	
NEXT	NeXT	
PGON	Intel Paragon	Uses native messages
PMAX	DECstation 3100, 5100	Ultrix
RS6K	IBM/RS6000	AIX 3.2
RT	IBM RT	
SGI	Silicon Graphics IRIS	IRIX 4.x
SGI5	Silicon Graphics IRIS	IRIX 5.x
SGIMP	SGI multiprocessor	Uses shared memory
SUN3	Sun 3	SunOS 4.2
SUN4	Sun 4, SPARCstation	SunOS 4.2
SUN4SOL2	Sun 4, SPARCstation	Solaris 2.x
SUNMP	SPARC multiprocessor	Solaris 2.x, uses shared memory
SYMM	Sequent Symmetry	
TITN	Stardent Titan	
U370	IBM 370	AIX
UVAX	DEC MicroVAX	

Table 1.3.1: PVM_ARCH names used in PVM 3

Table 1.3.1 lists the PVM_ARCH names and their corresponding architecture types that are supported in PVM 3.3.

The PVM source comes with directories and makefiles for most architectures you are likely to have.

CHAPTER 2: Starting PVM

Before we go over the steps to compile and run parallel PVM programs, it should be appropriate to start up PVM and configure a virtual machine. On any host on which PVM has been installed you can type

% pvm

and you should get back a PVM console prompt signifying that PVM is now running on this host. You can add hosts to your virtual machine by typing at the console prompt

pvm> add hostname

And you can delete hosts (except the one you are on) from your virtual machine by typing

pvm> delete hostname

If you get the message "Can't Start pvmd", then check the common startup problems section and try again.

To see what the present virtual machine looks like, you can type

pvm> conf

To see what PVM tasks are running on the virtual machine, you type

pvm> ps -a

Of course you don't have any tasks running yet; that's in the next section. If you type "quit" at the console prompt, the console will quit but your virtual machine and tasks will continue to run. At any Unix prompt on any host in the virtual machine, you can type

% pvm

and you will get the message "pvm already running" and the console prompt. When you are finished with the virtual machine, you should type

pvm> halt

This command kills any PVM tasks, shuts down the virtual machine, and exits the console. This is the recommended method to stop PVM because it makes sure that the virtual machine shuts down cleanly.

You should practice starting and stopping and adding hosts to PVM until you are comfortable with the PVM console. A full description of the PVM console and its many command options is given at the end of this chapter.

If you don't want to type in a bunch of host names each time, there is a hostfile option. You can list the hostnames in a file one per line and then type

% pvm hostfile

PVM will then add all the listed hosts simultaneously before the console prompt appears. Several options can be specified on a per-host basis in the hostfile . These are described at the end of this chapter for the user who wishes to customize his virtual machine for a particular application or environment.

There are other ways to start up PVM. The functions of the console and a performance monitor have been combined in a graphical user interface called XPVM , which is available precompiled on netlib. If XPVM has been installed at your site, then it can be used to start PVM. To start PVM with this X window interface, type;

% xpvm

The menu button labled "hosts" will pull down a list of hosts you can add. If you click on a hostname, it is added and an icon of the machine appears in an animation of the virtual machine. A host is deleted if you click on a hostname that is already in the virtual machine (see Figure 2.1). On startup XPVM reads the file $HOME/.xpvm_hosts, which is a list of hosts to display in this menu. Hosts without leading ``\&" are added all at once at startup.

The quit and halt buttons work just like the PVM console. If you quit XPVM and then restart it, XPVM will automatically display what the running virtual machine looks like. Practice starting and stopping and adding hosts with XPVM. If there are errors, they should appear in the window where you started XPVM.

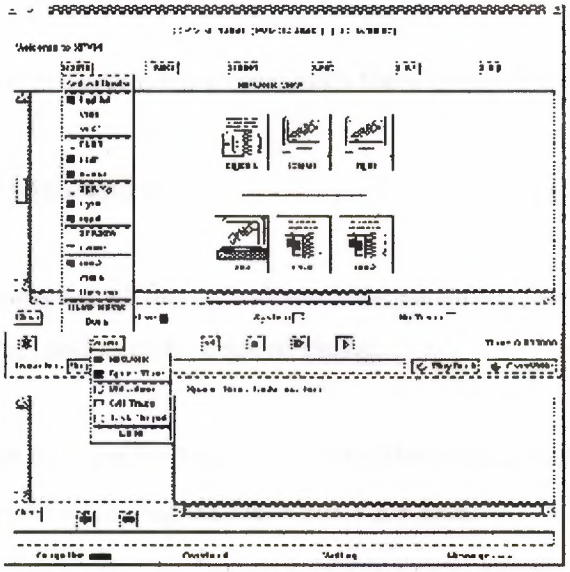


Figure 2.1: XPVM System Adding Hosts

2.1 PVM Console Details

The PVM console, called *pvm*, is a stand-alone PVM task that allows the user to interactively start, query, and modify the virtual machine. The console may be started and stopped multiple times on any of the hosts in the virtual machine without affecting PVM or any applications that may be running. When started, *pvm* determines whether PVM is already running; if it is not, *pvm* automatically executes *pvmd* on this host, passing *pvmd* the command line options and hostfile. Thus PVM need not be running to start the console.

pvm [-n<hostname>] [hostfile]

The -n option is useful for specifying an alternative name for the master *pvmd* (in case hostname doesn't match the IP address you want). Once PVM is started, the console prints the prompt;

pvm>

and accepts commands from standard input. The available commands are

add

followed by one or more host names, adds these hosts to the virtual machine.

alias

defines or lists command aliases.

conf

lists the configuration of the virtual machine including hostname, pvmd task ID, architecture type, and a relative speed rating.

delete

followed by one or more host names, deletes these hosts from the virtual machine. PVM processes still running on these hosts are lost.

echo

echo arguments.

halt

kills all PVM processes including console, and then shuts down PVM. All daemons exit.

help

can be used to get information about any of the interactive commands. Help may be followed by a command name that lists options and flags available for this command.

id

prints the console task id.

jobs

lists running jobs.

kill

can be used to terminate any PVM process.

mstat

shows the status of specified hosts.

ps -a

lists all processes currently on the virtual machine, their locations, their task id's, and their parents' task id's.

pstat

shows the status of a single PVM process.

quit

exits the console, leaving daemons and PVM jobs running.

reset

kills all PVM processes except consoles, and resets all the internal PVM tables and message queues. The daemons are left in an idle state.

setenv

displays or sets environment variables.

sig

followed by a signal number and TID, sends the signal to the task.

spawn

starts a PVM application. Options include the following:

-count

number of tasks; default is 1.

-host

spawn on host; default is any.

-ARCH

spawn of hosts of type ARCH.

-?

enable debugging.

->

redirect task output to console.

->file

redirect task output to file.

->>file

redirect task output append to file.

-@

trace job, display output on console

-@file

trace job, output to file

trace

sets or displays the trace event mask.

unalias

undefines command alias.

version

prints version of PVM being used.

The console reads *$HOME/.pvmrc* before reading commands from the tty, so you can do things like

```
alias ? help
alias h help
alias j jobs
setenv PVM_EXPORT DISPLAY
# print my id
echo new pvm shell
id
```

PVM supports the use of multiple consoles . It is possible to run a console on any host in an existing virtual machine and even multiple consoles on the same machine. It is also possible to start up a console in the middle of a PVM application and check on its progress.

2.2 Basic Programming Techniques

Developing applications for the PVM system-in a general sense, at least-follows the traditional paradigm for programming distributed-memory multiprocessors such as the nCUBE or the Intel family of multiprocessors. The basic techniques are similar both for the logistical aspects of programming and for algorithm development. Significant differences exist, however, in terms of (a) task management, especially issues concerning

dynamic process creation, naming, and addressing; (b) initialization phases prior to actual computation; (c) granularity choices; and (d) heterogeneity. In this chapter, we discuss the programming process for PVM and identify factors that may impact functionality and performance.

2.2.1 Common Parallel Programming Paradigms

Parallel computing using a system such as PVM may be approached from three fundamental viewpoints, based on the organization of the computing tasks. Within each, different workload allocation strategies are possible and will be discussed later in this chapter. The first and most common model for PVM applications can be termed crowd computing : a collection of closely related processes, typically executing the same code, perform computations on different portions of the workload, usually involving the periodic exchange of intermediate results. This paradigm can be further subdivided into two categories:

1. The master-slave (or host-node) model in which a separate "control" program termed the master is responsible for process spawning, initialization, collection and display of results, and perhaps timing of functions. The slave programs perform the actual computation involved; they either are allocated their workloads by the master (statically or dynamically) or perform the allocations themselves.
2. The node-only model where multiple instances of a single program execute, with one process (typically the one initiated manually) taking over the non computational responsibilities in addition to contributing to the computation itself.

The second model supported by PVM is termed a "tree" computation . In this scenario, processes are spawned (usually dynamically as the computation progresses) in a tree-like manner, thereby establishing a tree-like, parent-child relationship (as opposed to crowd computations where a star-like relationship exists). This paradigm, although less commonly used, is an extremely natural fit to applications where the total workload is not known *a priori*.

The third model, which we term "hybrid", can be thought of as a combination of the tree model and crowd model. Essentially, this paradigm possesses an arbitrary spawning structure: that is, at any point during application execution, the process relationship structure may resemble an arbitrary and changing graph.

These three classifications are made on the basis of process relationships, though they frequently also correspond to communication topologies. Nevertheless, in all three, it is possible for any process to interact and synchronize with any other. Further, as may be expected, the choice of model is application dependent and should be selected to best match the natural structure of the parallelized program.

2.2.2 Crowd Computation

Crowd computations typically involve three phases. The first is the initialization of the process group; in the case of node-only computations, dissemination of group information and problem parameters, as well as workload allocation, is typically done within this phase. The second phase is computation. The third phase is collection results and display of output; during this phase, the process group is disbanded or terminated.

The master-slave model is shown in Figure 2.2.2.1, using the well-known Mandelbrot set computation which is representative of the class of problems termed "embarrassingly" parallel . The computation itself involves applying a recursive function to a collection of points in the complex plane until the function values either reach a specific value or begin to diverge. Depending upon this condition, a graphical representation of each point in the plane is constructed. Essentially, since the function outcome depends only on the starting value of the point (and is independent of other points), the problem can be partitioned into completely independent portions, the algorithm applied to each, and partial results combined using simple combination schemes. However, this model permits dynamic load balancing, thereby allowing the processing elements to share the workload unevenly. In this and subsequent examples within this chapter, we only show a skeletal? form of the algorithms, and also take syntactic liberties with the PVM routines in the interest of

clarity. The control structure of the master-slave class of applications is shown in Figure 2.2.2.1.

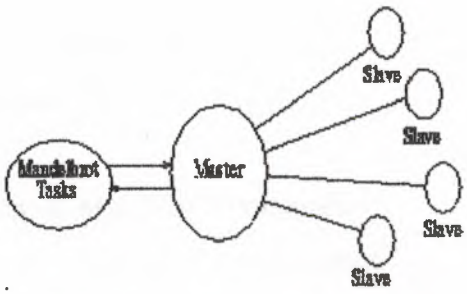


Figure 2.2.2.1: Master-Slave Paradigm

The master-slave example described above involves no communication among the slaves. Most crowd computations of any complexity do need to communicate among the computational processes to we illustrate the structure of such applications using a node-only example for matrix multiply using Cannon's algorithm. The matrix-multiply example, shown pictorially in Figure 2.2.2.2. multiplies matrix sub-blocks locally, and uses row-wise multicast of matrix A sub-blocks in conjunction with column-wise shifts of matrix B subblocks.

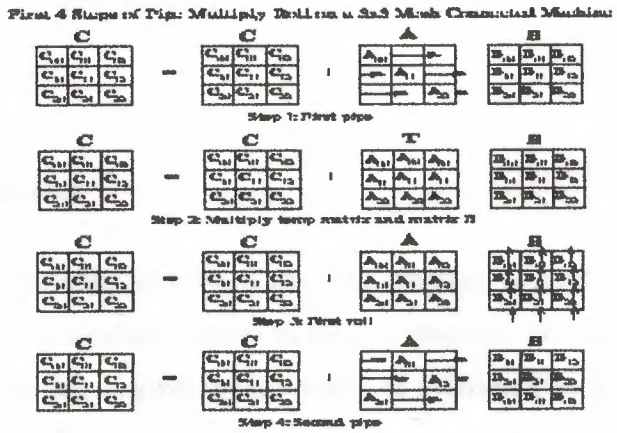


Figure 2.2.2.2: General Crowd Computation

{Matrix Multiplication Using Pipe-Multiply-Roll Algorithm}

{Processor 0 starts up other processes}

if (<my processor number> = 0) then

for i := 1 to MeshDimension*MeshDimension

```
    pvm_spawn(<component name>, . .)
  endfor
endif
forall processors Pij, 0 <= i,j < MeshDimension
  for k := 0 to MeshDimension-1
    {Pipe.}
    if myrow = (mycolumn+k) mod MeshDimension
      {Send A to all Pxy, x = myrow, y <> mycolumn}
      pvm_mcast((Pxy, x = myrow, y <> mycolumn),999)
    else
      pvm_recv(999)  {Receive A}
    endif
      {Multiply. Running totals maintained in C.}
    Multiply(A,B,C)
     {Roll.}
    {Send B to Pxy, x = myrow-1, y = mycolumn}
    pvm_send((Pxy, x = myrow-1, y = mycolumn),888)
    pvm_recv(888)     {Receive B}
  endfor
endfor
```

2.2.3 Tree Computations

Tree computations typically exhibit a tree-like process control structure which also conforms to the communication pattern in many instances. To illustrate this model, we consider a parallel sorting algorithm that works as follows. One process (the manually started process in PVM) possesses (inputs or generates) the list to be sorted. It then spawns a second process and sends it half the list. At this point, there are two processes each of which spawns a process and sends them one-half of their already halved lists. This continues until a tree of appropriate depth is constructed. Each process then independently sorts its portion of the list, and a merge phase follows where sorted sublists are transmitted upwards along the tree edges, with intermediate merges being done at

each node. This algorithm is illustrative of a tree computation in which the workload is known in advance; a diagram depicting the process is given in Figure 2.2.3.1; an algorithmic outline is given below.

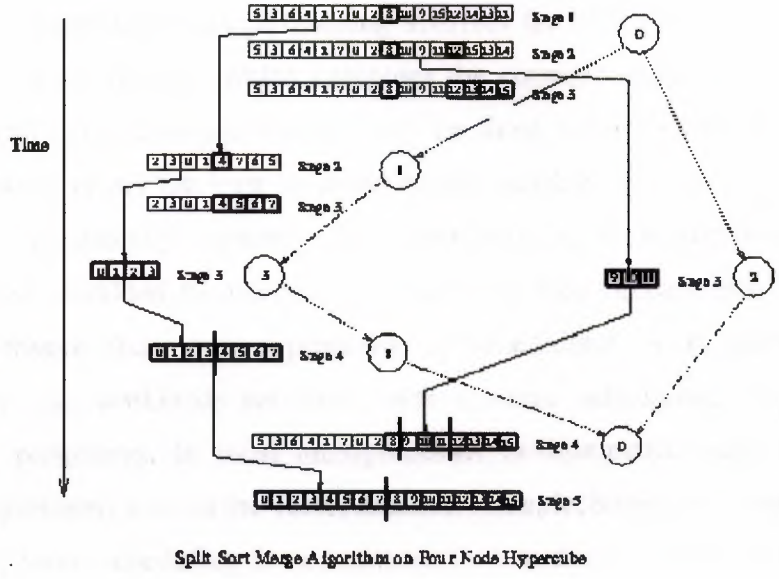


Figure 2.2.3.1: Tree-Computation Example

2.3 Workload Allocation

The common parallel programming paradigms with respect to process structure, and we outlined representative examples in the context of the PVM system. This section addresses the issue of workload allocation, subsequent to establishing process structure, and describe some common paradigms that are used in distributed-memory parallel computing. Two general methodologies are commonly used. The first, termed *data decomposition* or partitioning, assumes that the overall problem involves applying computational operations or transformations on one or more data structures and, further, that these data structures may be divided and operated upon. The second, called *function decomposition*, divides the work based on different operations or functions. The PVM computing model supports both function decomposition and data decomposition.

2.3.1 Data Decomposition

As a simple example of data decomposition, consider the addition of two vectors, A[1..N] and B[1..N], to produce the result vector, C[1..N]. If we assume that P processes are working on this problem, data partitioning involves the allocation of N/P elements of each vector to each process, which computes the corresponding N/P elements of the resulting vector. This data partitioning may be done either "statically", where each process knows *a priori* (at least in terms of the variables N and P) its share of the workload, or "dynamically", where a control process (e.g., the master process) allocates subunits of the workload to processes as and when they become free. The principal difference between these two approaches is "scheduling". With static scheduling, individual process workloads are fixed; with dynamic scheduling, they vary as the computation progresses. In most multiprocessor environments, static scheduling is effective for problems such as the vector addition example; however, in the general PVM environment, static scheduling is not necessarily beneficial. The reason is that PVM environments based on networked clusters are susceptible to external influences; therefore, a statically scheduled, data-partitioned problem might encounter one or more processes that complete their portion of the workload much faster or much slower than the others. This situation could also arise when the machines in a PVM system are heterogeneous, possessing varying CPU speeds and different memory and other system attributes.

In a real execution of even this trivial vector addition problem, an issue that cannot be ignored is input and output. In other words, how do the processes described above receive their workloads, and what do they do with the result vectors? The answer to these questions depends on the application and the circumstances of a particular run, but in general:

1. Individual processes generate their own data internally, for example, using random numbers or statically known values. This is possible only in very special situations or for program testing purposes.

2. Individual processes independently input their data subsets from external devices. This method is meaningful in many cases, but possible only when parallel I/O facilities are supported.
3. A controlling process sends individual data subsets to each process. This is the most common scenario, especially when parallel I/O facilities do not exist. Further, this method is also appropriate when input data subsets are derived from a previous computation within the same application.

The third method of allocating individual workloads is also consistent with dynamic scheduling in applications where inter process interactions during computations are rare or nonexistent. However, nontrivial algorithms generally require intermediate exchanges of data values, and therefore only the initial assignment of data partitions can be accomplished by these schemes. In order to multiply two matrices A and B, a group of processes is first spawned, using the master-slave or node-only paradigm. This set of processes is considered to form a mesh; the matrices to be multiplied are divided into sub-blocks, also forming a mesh. Each sub-block of the A and B matrices is placed on the corresponding process, by utilizing one of the data decomposition and workload allocation strategies listed above. During computation, sub-blocks need to be forwarded or exchanged between processes, thereby transforming the original allocation map, as shown in the figure. At the end of the computation, however, result matrix sub-blocks are situated on the individual processes, in conformance with their respective positions on the process grid, and consistent with a data partitioned map of the resulting matrix C. The foregoing discussion illustrates the basics of data decomposition. In a later chapter, example programs highlighting details of this approach will be presented .

2.3.2 Function Decomposition

Parallelism in distributed-memory environments such as PVM may also be achieved by partitioning the overall workload in terms of different operations. The most obvious example of this form of decomposition is with respect to the three stages of typical program execution, namely, input, processing, and result output. In function decomposition, such an application may consist of three separate and distinct programs,

each one dedicated to one of the three phases. Parallelism is obtained by concurrently executing the three programs and by establishing a "pipeline" (continuous or quantized) between them. Data parallelism may also exist within each phase. An example is shown in Figure 2.3.1, where distinct functions are realized as PVM components, with multiple instances within each component implementing portions of different data partitioned algorithms.

Although the concept of function decomposition is illustrated by the trivial example above, the term is generally used to signify partitioning and workload allocation by function *within* the computational phase. Typically, application computations contain several different sub-algorithms-sometimes on the same data (the MPSD or multiple-program single-data scenario), sometimes in a pipelined sequence of transformations, and sometimes exhibiting an unstructured pattern of exchanges. We illustrate the general functional decomposition paradigm by considering the hypothetical simulation of an aircraft consisting of multiple interrelated and interacting, functionally decomposed sub-algorithms. A diagram providing an overview of this example is shown in Figure 2.3.1.

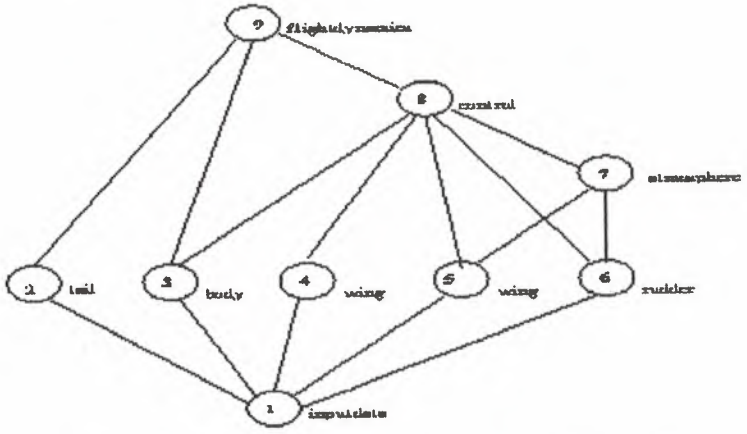


Figure 2.3.1: Function Decomposition Example

In the figure 2.3.1, each node or circle in the "graph" represents a functionally decomposed piece of the application. The input function distributes the particular problem parameters to the different functions 2 through 6, after spawning processes corresponding to distinct programs implementing each of the application sub-algorithms. The same data may be sent to multiple functions (e.g., as in the case of the two *wing*

functions), or data appropriate for the given function alone may be delivered. After performing some amount of computations these functions deliver intermediate or final results to functions 7, 8, and 9 that may have been spawned at the beginning of the computation or as results become available. The diagram indicates the primary concept of decomposing applications by function, as well as control and data dependency relationships. Parallelism is achieved in two respects-by the concurrent and independent execution of modules as in functions 2 through 6, and by the simultaneous, pipelined, execution of modules in a dependency chain, as, for example, in functions 1, 6, 8, and 9 .

CHAPTER 3: PVM Application

3.1 Core Features of PVM

The routines supplied by PVM include adding and deleting hosts from the virtual machine, enabling a user to process to register-to-leave a collection of cooperating processes, to synchronize with and send signals to and other PVM tasks, to initiate and terminate tasks, and routines to obtain information about the virtual machine configuration and active tasks. Synchronization of tasks is achieved by sending a signal from the host to another task of different host or by using barriers. Another synchronization method would be notifying a set of tasks of an event occurrence by sending them a message with the user-specified tag that the application can check for. The notification of events includes the exiting of a task, the deletion of a host, and addition of a new host.

The packing and sending of messages between tasks is provided by PVM routines. The communication routines include an asynchronous send to a single task, and multicast to list of tasks. Messages are moved over the underlying network by PVM using UDP, TCP, or some other high speed interconnects available between the two hosts. A routine can be called to return information about received messages such as the source, tag, and size of the data. The message buffers are allocated dynamically, permitting messages limited in size only by native machine parameters. There are, however, routines for creating and managing multiple send and receive buffers. The user can thus write PVM math libraries and graphical interfaces that can be called inside other PVM applications.

A process in PVM can belong to multiple groups, and groups can change dynamically at any time of computation. Dynamic process groups are layered above the core PVM routines. The PVM routines are provided for tasks to join and leave the named group. Members of the group are numbered from zero to the number of group members-1. Tasks can also query for information about other group members.

Messages designated for an external host are routed via the user's PVM daemon on the multiprocessor, while messages between two nodes use its native message-passing routines. The data movement can be implemented with shared buffer tool and lock primitives on shared-memory-systems. The daemon that manages the allocation and reallocation of nodes on the multiprocessor is the construct of MPP system of PVM. The second part of MPP is a specialized libpvm library for this architecture that contains the fast routing calls between nodes of this host.

The daemon *pvmd* keeps track of the entire task management. The first *pvmd* is the master and the hoster process becomes the slave. The master *pvmd* is used to startup slave daemons on other hosts. The hoster process starts automatically and is running as a task on the virtual machine. The PVM startup protocol is given in figure 3.1.

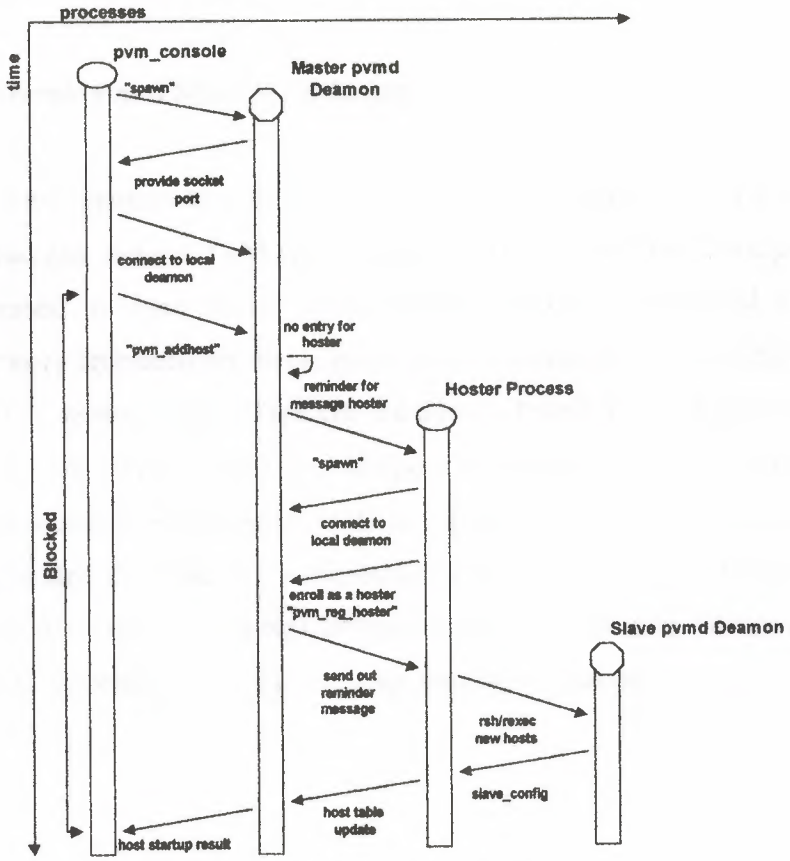


Figure 3.1: PVM Startup Protocol

3.1.1 Data Transfer and Barrier Synchronization

Inter-process communication via message passing is one of the basic facilities supported by PVM. Since the physical location of processes is transparent to user programs, a pair identifies message destinations. Furthermore, owing to the heterogeneous nature of the underlying hardware that PVM executes on, it is necessary for user programs to send and receive typed data in a machine independent form.

Synchronization via barriers is a common operation in many applications. Using PVM, barrier synchronization is accomplished using the *barrier* construct. The PVM system attempts to detect and correct barrier deadlocks by notifying invoking processes if some instances of a component terminate before they reach a barrier, and a residual number of instances of a component terminate before they reach the barrier, and the residual number of instances cannot form a barrier.

3.1.2 Shared Memory and Mutual Exclusion

The use of shared memory to synchronize and communicate between processes is a convenient paradigm, and the PVM system provides such an interface for algorithms that are best expressed in these terms. Since shared memory is emulated by PVM on distributed memory architectures, some performance degradation is inevitable when the granularity of the access is fine. This can be demonstrated by the following 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 sufficiently large in size. Since, PVM permits each worker in the pool to run on different architectures, the individual worker components may be written to internally use either message passing or shared memory.

3.2 Implementation

The PVM support software executed as a user-level process on each host in the participant pool. Additions or deletions are possible during the 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 on a host is responsible for all application component process executing on that host. Control is completely distributed 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 Operating System dependent mechanism.

The PVM system is composed of two parts. The *pvmd3,* which is the first part called the daemon, resides on all of the computers making the virtual machine. This daemon is designed to run applications. This is achieved by executing *pvmd3* on each of the computers making up the user-defined virtual machine. The PVM application can then be started from any host's machine's prompt.

The second part of the systems is a library of PVM interface routines, which consists of callable routines for message-passing, spawning processes, coordinating tasks, and modifying the virtual machine. Application programs developed must be linked to this library.

The PVM system assumes only that unreliable, unsequnced, point-to-point data transfer 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.

The *pvmd* processes across the network communicate using the UDP datagrams. The "well known port" approach is used for addressing where all incoming messages are

received by *pvmd* processes on a predetermined port number. The first communication instance between any two entries is routed through the *pvmd* process on the source and destination machines. Location and port number information is appended to this exchange where the PVM routines that implement *send* and *receive* cache this information, thus enabling direct communication for subsequent changes. Local user processes communicate with *pvmd* using the most efficient machine dependent mechanism.

3.2.1 Point-to-Point Data Transfer

The *pvmd* processes use a positive acknowledgement scheme and additional header that contains sequence numbers as well as fragmentation and reassemble information to achieve reliable and sequenced point-to-point communications. Unacknowledged transmissions are retried a parameterized number of times after which recipient process or processor is presumed to be uncooperative. 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 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.

3.3 Performance Considerations of PVM

PVM and virtual machines usually operate in general-purpose networked environments, where no dedication of the CPU of the individual machines nor the interconnection network are considered. Due to variations in network traffic delay the raw performance or speedup of a given application is hard to measure. To improve performance on dedicated networks, the traffic rate should be minimal. Even in a dedicated networked environment, the above assumption is true since operating system activity, window and file system

overheads, and administrative network traffic can contribute to deviated measurements. The network computing systems behave in a manner that is reasonable, predictable if these factors are ignored. Since parallelism granularity is at the process level, most of the focus is on communication overhead; CPU optimizations can be approached independently using traditional methods.

3.3.1 Raw Communications Performance of PVM

The performance considerations of PVM can be analyzed with data transfer costs, The time required for processes to exchange messages is dependent on several factors, including the host machines, network speeds, and most predominantly, the message size. Apart from point-to-point data transfer, group communication facilities are also an important measure of communications performance.

The communication throughput approaching the medium capacity can be achieved in PVM, provided large messages are transferred. Under situations involving multiple, simultaneous message passing, high performance of the *aggregate* bandwidth of the medium is utilizable by PVM application processes. The factor that is difficult to optimize is latency, implying poor efficiency and speedup when message exchanges are short.

3.4 Scientific Computing

Many important scientific, industrial, and mechanical problems are being solved using PVM over a cluster of workstations. The driving force behind the initial popularity of PVM was to get very good price performance ratio. In general a cluster of about 10 high performance workstations is potentially capable of solving a problem as fast as a supercomputer costing much more. This attractive popularity has inspired third parity software vendors to develop tools, which assist in paralleling, a user's application by placing PVM calls. High degree of portability and a straightforward, robust interface that is well suited for scientific application development are other motivations for the increasing use of PVM.

3.5 Beginning Programming

There are a few structures that are common to all PVM programs written in C. Every PVM program should include the PVM header file. This contains needed information about the PVM programming interface. This is done by putting

#include "pvm3.h" at the beginning of a C program or #include "fpvm3.h" in a Fortran program.

The first PVM function called by a program, commonly *info=pvm_mytid(),* enrolls the process in PVM. info is an integer returned by the function. Like all PVM functions, *pvm_mytid()* will return a negative number if an error occurs. Programs should check for these errors, and respond appropriately. When a PVM program is done, it should call *pvm_exit().*

In order to write a parallel program, tasks must be executed on different processors. This is accomplished by calling *pvm_spawn().* The man pages explain this function in detail. Here is an example of a typical call to *pvm_spawn().*

numt=pvm_spawn("my_task", NULL, PvmTaskDefault, 0, n_task, tids)

This spawns n_task copies of the program "my_task" on the computers that PVM chooses. The actual number of tasks started is returned to numt. The task id of each task that is spawned is returned in the integer array TIDs. Every PVM process has a unique integer that identifies it.

PVM has many useful information functions built into it. Some examples are *pvm_parent(), pvm_config(), pvm_tasks(),* etc... These can give your programs access to the types of information available at the PVM console.

3.5.1 Compiling and Running Your Program

In order to run your programs, you must compile them. On my machine, `cc -L~/pvm3/lib/ALPHA foo.c -lpvm3 -o foo` will compile a program called foo.c. You will have to change the name ALPHA to the architecture name of your computer. After compiling, you must put the executable file in the directory ~/pvm3/bin/ARCH. Also, you need to compile the program separately for every architecture in your virtual machine. If you use dynamic groups, you must also add -lgpvm3 to the compile command.

Your executable file can then be run. To do this, you should first run PVM. After PVM is running, your executable may be run from the Unix command line, like any other program.

3.5.2 Communication Between Tasks

Once you have learned how to spawn tasks and compile your programs, you are ready to do some actual parallel programming. To do this, you must learn how to let different tasks communicate with each other. In PVM, task-to-task communication is done with message passing.

When you are ready to send a message from task A to task B, task A must first call *pvm_initsend()*. This clears the default send buffer and specifies the message encoding. *bufid=pvm_initsend(PvmDataDefault)* is a typical use of this. After initialization, the sending task must then pack all of the data it wishes to send into the sending buffer. This is done with calls to the *pvm_pack()* family of functions. *pvm_packf()* is a printf-like function for packing multiple types of data. There are also functions for packing arrays of a single type of data, such as *pvm_pkdbl()* for packing doubles. The man pages have all the details on the different packing functions under *pvm_pack()*.

After the data have been packed into the sending buffer, the message is ready to be sent. This is accomplished with a call to *pvm_send()*. *info=pvm_send(tid, msgtag)* will send

the data in the sending buffer to the process with the task id of tid. It tags the message with the integer value msgtag. A message tag is useful for telling the receiving task what kind of data it is receiving. For example, a message tag of 5 might mean add the numbers in the message, while a tag of 10 might mean multiply them. *pvm_mcast()* is a similar function. It does the same thing as *pvm_send(),* except it takes an array of tids instead of just one. This is useful when you want to send the same message to a set of tasks.

The receiving task makes a call to *pvm_recv()* to receive a message. *bufid=pvm_recv(tid, msgtag)* will wait for a message from task tid with tag msgtag to arrive, and will receive it when it does. A -1 can be specified for either the tid or msgtag, and will match anything. PVM 3.3 adds *pvm_trecv(),* which has the added ability to time out after a specified length of time. *pvm_nrecv()* can also be useful. This does a non-blocking receive--if there is a suitable message it is received, but if there isn't, the task does not wait. Sometimes *pvm_probe()* can be helpful as well. This will tell if a message has arrived, but takes no further action.

When a task has received a message, it must unpack the data from the receiving buffer. *pvm_unpack()* accomplishes this in the same manner that *pvm_pack()* uses to pack the data in. All data must be unpacked in exactly the same order as it was packed. Note that C structures must be packed element by element.
PVM 3.3 adds the function *pvm_psend()* for convenience. This packs and sends data in one call. PVM also includes routines for manipulating send and receive buffers directly. See the man pages on *pvm_mkbuff()* and *pvm_freebuf()* if you are interested in this.

3.6 Dynamic Process Groups

Dynamic process groups can be used when a set of tasks performs the either the same function or a group of closely related functions. Users are able to give a name to a set of PVM processes, which are all given a group instance number in addition to their tid.

When a task calls *inum=pvm_joingroup("group_name")*, it will be added to the group "group_name". If no such group exists, it will be created. The group instance number is returned in inum. This will be 0 for the first group member, and the lowest available integer for the rest of the group members. A task may belong to more than one group at a time. To leave a group, a tasks makes a call to *pvm_lvgroup()*.

There are group information functions. *pvm_getinst()*, *pvm_gettid()* , and *pvm_gsize()* return a process's group instance, tid, and group size, respectively. Other useful group functions are *pvm_bcast()* and *pvm_barrier()*. *pvm_bcast()* is very similar to *pvm_mcast()*, but instead of sending a message to an array of TID, it sends it to all members of a group. *pvm_barrier()* is used for synchronization. A task that calls *pvm_barrier()* will stop until all the members of its group call *pvm_barrier()* as well.

There are other group functions. *pvm_gather()*, *pvm_scatter()*, and *pvm_reduce()* are some examples of group functions you may find useful.

3.7 Load Balancing

Load balancing is very important for applications. Making sure that each host is doing its fair share of work can be a real performance enhancer.

The simplest method is *static* load balancing. In this method, the problem is divided up and tasks assigned to processors only once. The partitioning may occur before the job starts, or as an early step in the application. The size and number of tasks can be varied depending on the processing power of a given machine. On a lightly loaded network, this scheme can be quite effective.

When computational loads vary, a more sophisticated *dynamic* method of load balancing is required. The most popular method is called the *Pool of Tasks* paradigm. This is typically implemented as a master/slave program where the master manages a set of tasks. It sends slaves jobs to do as they become idle. This method is used in the sample

program xep supplied with the distribution. This method is not suited for applications which require task to task communication, since tasks will start and stop at arbitrary times. In this case, a third method may be used. At some predetermined time, all the processes stop; the work loads are then reexamined and redistributed as needed. Variations of these methods are possible for specific applications.

3.8 Distributed Computing

Advantages:

1. Using existing hardware, keeps costs low
2. Performance can be optimised by assigning each individual task to the most appropriate architecture
3. Exploitation of the heterogeneous nature of a computation ie provides access to different types of processors for those parts of an application that can only run on a certain platform
4. Virtual computer resources can grow in stages and take advantage of the latest computational and network technologies
5. Program development can be enhanced by using a familiar environment ie editors, compilers, debuggers that are available on individual machines
6. Individual computers and workstations are usually stable and substantial expertise in their use should be available
7. User-level or program-level fault tolerance can be implemented with little effort either in the application or in the underlying operating system
8. Facilitates collaborative work

3.9 Message Buffers

If the user is using only a single send buffer (and this is the typical case) then *pvm_initsend()* is the only required buffer routine. It is called before packing a new message into the buffer. The routine *pvm_initsend* clears the send buffer and creates a new one for packing a new message. The encoding scheme used for this packing is set by

encoding. The new buffer identifier is returned in bufid. The encoding options are as follows:

The following message buffer routines are required only if the user wishes to manage multiple message buffers inside an application. Multiple message buffers are not required for most message passing between processes. In PVM 3 there is one *active* send buffer and one *active* receive buffer per process at any given moment. The developer may create any number of message buffers and switch between them for the packing and sending of data. The packing, sending, receiving, and unpacking routines affect only the *active* buffers.

int bufid = pvm_mkbuf(int encoding)

call pvmfmkbuf(encoding, bufid)

The routine *pvm_mkbuf* creates a new empty send buffer and specifies the encoding method used for packing messages. It returns a buffer identifier bufid.

int info = pvm_freebuf(int bufid)

call pvmffreebuf(bufid, info)

The routine *pvm_freebuf()* disposes of the buffer with identifier bufid. This should be done after a message has been sent and is no longer needed. Call *pvm_mkbuf()* to create a buffer for a new message if required. Neither of these calls is required when using *pvm_initsend(),* which performs these functions for the user.

int bufid = pvm_getsbuf(void)

call pvmfgetsbuf(bufid)

int bufid = pvm_getrbuf(void)

call pvmfgetrbuf(bufid)

pvm_getsbuf() returns the active send buffer identifier. *pvm_getrbuf()* returns the active receive buffer identifier.

int oldbuf = pvm_setsbuf(int bufid)

call pvmfsetrbuf(bufid, oldbuf)

int oldbuf = pvm_setrbuf(int bufid)

call pvmfsetrbuf(bufid, oldbuf)

These routines set the active send (or receive) buffer to bufid, save the state of the previous buffer, and return the previous active buffer identifier oldbuf.

If bufid is set to 0 in *pvm_setsbuf()* or *pvm_setrbuf()*, then the present buffer is saved and there is no active buffer. This feature can be used to save the present state of an application's messages so that a math library or graphical interface which also uses PVM messages will not interfere with the state of the application's buffers. After they complete, the application's buffers can be reset to active.

It is possible to forward messages without repacking them by using the message buffer routines. This is illustrated by the following fragment.

```
bufid = pvm_recv( src, tag );
oldid = pvm_setsbuf( bufid );
info  = pvm_send( dst, tag );
info  = pvm_freebuf( oldid );
```

3.10 Matrix Multiplication

One of the PVM applications that would be an impact on MPP systems is matrix multiplication. Since matrix multiplication consumes CPU time and memory, it's one of the basic examples to be implemented under PVM to validate performance measures. The general algorithm of matrix multiplication is described below, and implemented with parallel running machines on a local network environment.

The tasks in PVM for matrix multiplication is thought to be two dimensional conceptual torus, for which there is no restriction on which tasks may communicate with each other. Groups are formed, **matrix_group** to enumerate the tasks. Group ids are used to map tasks. The first task to join the group is assigned an id of zero. The first task spawns the other tasks and sends the parameters for matrix multiply to those tasks. When all the

parameters are transmitted, and all the tasks have been spawned, a barrier program, **pvm_barrier()**, is executed to ensure that all tasks have joined the group. Then all the task ids are stored to array **myrow** by calculating the group ids for all the tasks and asking PVM for the entire task id for the corresponding group id.

The program calculates $C=AB$, where C, A, and B are square matrices. There are $m \times m$ Tasks required finding the solution. Each sub-block of resulting matrix will be calculated by each task. The matrices A and B are also stored and distributed over $m \times m$ tasks. The row and column of block C is calculated based on the value of the group id. The group ids range from 0 to $m-1$.

In order to perform matrix multiplication's, the tasks on the diagonal multicast their block of A to other tasks in their row. The shifting to the B blocks vertically occurs after the sub-blocks have been multiplied and added to the C block.

The blocks to be computed are first initialized by calling **InitBlock()**, where A is assigned random variables, B to identity matrix, and C to zero. Verification at the end of the program is performed by checking A=C.

3.11 Simulation and Test Results

To demonstrate the matrix multiplication program, the PVM test bed was set up. The PVM software along with remote shell (rsh) for adding hosts was loaded on each machine under Windows NT Operating System. All the tests for matrix multiplication were performed under a local network domain.

The PVM module is loaded on each of the PC workstation modules, and remote *rshd* software running on every machine interconnected by the local network domain. Each PVM software is installed on every machine as a server module to corporate the task management between the processes. The program is modified such that spawning occurs on between active tasks.

To verify test simulation results, various changes have been performed on the program provided within the PVM book. The program provided can be modified to apply for any two type of matrix multiplication with ease. Thus, it can be used as a reference tool in general for two-matrix multiplication being square matrices of different block size lengths, which would give reliable test results under PVM. The simulation test results yield to the following conclusions.

1. The response time of the PVM system would give less performance measure when host machines are interconnected by the network compared to the theoretical results. This is due to delay in network response time, which reduces the computational power of PVM due to message passing delays in the network. Different measurements have been taken of the matrix multiplication program, and the best results of the response time were obtained when the network was idle.
2. The response time of a single host machine running under PVM with increasing the number of block sizes of matrices increases exponentially with time. From 100 to 600 block size, the response time is slower compared to response time between 600 to 1200.

You can see Matrix Multiplication in PVM in Appendix A.

Appendix A:

```
/*************************************************************************
*******/
/* This program multiplies two square Matrices A and B, resulting to C      */
/* under PVM. Matrix A is assigned using random variables, and B to identity. */
/* The resulting C matrix = A.                                              */
/* The program also measures the response time taken in calculating C.      */
/*************************************************************************
*******/
/*                                                                          */
```

```
/* The measurement is a standard ping-pong with roundtrip time measured
*/
/* from the originating sender.
                        */
/*
                                                        */
/*****************************************************************/
/* Defines the prototypes for the PVM Library */
#include <pvm3.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
/* Maximum Number of Children This Program Will Spawn */
#define     MAXNTIDS  1500
#define  MAXROW       10
/* Message Tags */
#define             ATAG 2
#define     BTAG 3
#define             DIMTAG      5
#define     group  "mmult"
void
InitBlock(float *a, float *b, float *c, int blk, int row, int col)
{
        int     len, ind;
  int   i,j;
  srand(pvm_mytid());
  len = blk*blk;
  for (ind = 0; ind < len; ind++)
        {a[ind]=(float)(rand()%1000)/100.0;c[ind] = 0.0;}
  for (i = 0; i < blk; i++) {
        for (j = 0; j < blk; j++) {
```

```
        if (row == col)
          b[j*blk+i] = (i==j)? 1.0 : 0.0;
        else
          b[j*blk+i] = 0.0;
      }
    }
}

void
BlockMult(float *c, float *a, float *b, int blk)
{
        int     i,j,k;
  for (i = 0; i < blk; i++)
        for (j = 0; j < blk; j++)
        for (k = 0; k < blk; k++)
        c[i*blk+j] += (a[i*blk+k] * b[k*blk+j]);
}
int
main(int argc, char* argv[])
{
        /* Start PVM Clock Timer */
        clock_t start, end;

        /* Number of Tasks to Spawn, Using 2 as Default */
        int     ntask = 2;

  /* Return Code From PVM Calls */
        int     info;

  /* My Task and Group id */
        int     mytid, mygid;
```

```
/* Children Task id Array */
        int     child[MAXNTIDS-1];
        int     i, m, blksize;

/* Array of the TIDS in My Row */
        int     myrow[MAXROW];
        float   *a, *b, *c, *atmp;
        int     row, col, up, down;

        FILE *ff = fopen("d:\\pvmtmp\\matrix\\mymatrix\\mmult.out", "w");
        printf("Please Wait While PVM Starts Matrix Multiplication...\n");

/* Find out Task id Number */
mytid = pvm_mytid();
//pvm_setopt(PvmRoute, PvmRouteDirect);
/* Check For Error */
if (mytid < 0) {

        /* Print Out The ERROR */
  pvm_perror(argv[0]);

  /* Exit the Program */
  return -1;
  }
        start = clock();
/* Join the MMULT Group */
mygid = pvm_joingroup("mmult");
if (mygid < 0) {
        pvm_perror(argv[0]); pvm_exit(); return -1;
   }
```

```
/* If My Group id is 0 then I Must Spawn the Other Tasks */
if (mygid == 0){

    /* Find Out How Many Tasks to Spawn */
  if (argc == 3) {
      m = atoi(argv[1]);
    blksize = atoi(argv[2]);
   }
  if (argc < 3) {
      fprintf(ff, "usage:mmult m blk\n");
    pvm_lvgroup("mmult"); pvm_exit(); return -1;
    }

  /* Make Sure ntast is Leagal */
  ntask = m*m;
  if ((ntask < 1) || (ntask >= MAXNTIDS)) {
      fprintf(ff, "ntask = %d not valid.\n", ntask);
    pvm_lvgroup("mmult"); pvm_exit(); return -1;
    }

  /* No Need To Spawn if There is Only One Task */
  if (ntask == 1) goto barrier;

  /* Spawn The Child Tasks */
  info = pvm_spawn("mmult", (char**)0, PvmTaskDefault, (char*)0,
  ntask-1, child);
            /* Make Sure Spawn Succeeded */
  if (info !=ntask-1) {
      pvm_lvgroup("mmult"); pvm_exit(); return -1;
                }
  /* Send The Matrix Dimension */
```

```
  pvm_initsend(PvmDataDefault);
  pvm_pkint(&m, 1, 1);
  pvm_pkint(&blksize, 1, 1);
  pvm_mcast(child, ntask-1, DIMTAG);
  }
else {
      /* Receive Matrix Dimension */
  pvm_recv(pvm_gettid("mmult", 0), DIMTAG);
  pvm_upkint(&m, 1, 1);
  pvm_upkint(&blksize, 1, 1);
  ntask = m*m;
  }
  /* Make Sure All Tasks Have Joined The Group */
barrier:
      info = pvm_barrier("mmult", ntask);
  if (info < 0) pvm_perror(argv[0]);

  /* Find the tids In My Row */
  for (i = 0; i < m; i++)
      myrow[i] = pvm_gettid("mmult", (mygid/m)*m +i);
  /* Allocate The Memory For The Local Blocks */
  a = (float*)malloc(sizeof(float)*blksize*blksize);
  b = (float*)malloc(sizeof(float)*blksize*blksize);
  c = (float*)malloc(sizeof(float)*blksize*blksize);
  atmp = (float*)malloc(sizeof(float)*blksize*blksize);
  /* Check for the Valid Pointers */
  if (!(a && b && c && atmp)) {
      fprintf(ff, "%s:out of memory!\n", argv[0]);
    free(a); free(b); free(c); free(atmp);
    pvm_lvgroup("mmult"); pvm_exit(); return -1;
    }
```

```
/* Find My Block's row and column */
row = mygid/m; col = mygid % m;

/*Calculate the Neighbour's Above and Below */
up = pvm_gettid("mmult", ((row)?(row-1):(m-1))*m+col);
down = pvm_gettid("mmult", ((row == (m-1))?col:(row+1)*m+col));

    /* Initialize The Blocks */
InitBlock(a, b, c, blksize, row, col);

/* Do The Matrix Multiply */
for (i = 0; i < m; i++) {

    /*mcast the Block of Matrix A */
  if (col == (row +i)%m) {
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(a, blksize*blksize, 1);
    pvm_mcast(myrow, m, (i+1)*ATAG);
    BlockMult(c, a, b, blksize);
    }
  else {
    pvm_recv(pvm_gettid("mmult", row*m + (row + i)%m), (i+1)*ATAG);
    pvm_upkfloat(atmp, blksize*blksize, 1);
    BlockMult(c, atmp, b, blksize);
    }
    /*Rotate the Colums of B */
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(b, blksize*blksize, 1);
    pvm_send(up, (i+1)*BTAG);
    pvm_recv(down, (i+1)*BTAG);
```

```
        pvm_upkfloat(b, blksize*blksize, 1);
        }
        /* End PVM Timer */
                end = clock();
                printf("\nThe Response Time of PVM was: %f ", (float)(end - start) /
(float)CLK_TCK);
                printf("sec \n");

        /* check it */
      for (i = 0; i < blksize*blksize; i++)
        if (a[i] != c[i])
                printf("Error a[%d] (%g) !=c[%d] (%g) \n", i, a[i], i, c[i]);
      printf("PVM Matrix Multiplication Completed and Checked...\n");
      free(a); free(b); free(c); free(atmp);
      pvm_lvgroup("mmult");
    pvm_exit();
        return 0;
}
```

CONCLUSION

The primary motivation for the PVM are derived from the existing and anticipated need for general, flexible, and inexpensive concurrent programming environment. The PVM system provides heterogeneous framework environment that can executed on existing hardware bases, with the benefits of a procedural programming interface and straightforward construct for access to various resources. Another feature of PVM is its generality in supporting both shared-memory and message passing paradigms on heterogeneous collection of machines. The framework provided by PVM enables interaction between application components and machine architectures that are normally incompatible. Furthermore, PVM features have been built such that large and complex parallel system will require error indication and failure detection capabilities. From the performance point of view, PVM has been proven to be acceptable even when application with high communication to communication ratio, although it's primary intent is to support applications with much larger grainsize and less interaction. In situations where PVM gains more importance is the ability of PVM to utilize resource that already exist and would be a 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 many applications areas in which PVM needs to be improved. Some future work application are discussed. A very valuable addition would be to a supplemental toolkit that assisted in algorithm partitioning and scheduling, as well as support for automatic compilation of different object modules for different architectures. From the system implementation point of view, it will be evident that the data transfer, broadcast, and mutual exclusion protocols are the most crucial primitives, and work in progress to optimize these. From the application point of view, certain additional features might be desirable such as the ability to coalesce emulated and real shared memory, and to dynamically optimize message passing, locking, and design of a graphical interface for the specification of component execution order and interactions, as well as debugging and execution history trace facilities.

REFERENCES

[1] Lecture Notes of Com 442 given by Mr Halil ADAHAN, 2002

[2] www.PVM.com, The PVM main page, 2003

[3] Introduction to PVM,

[4] www.google.com

[5] www.yahoo.com

[6] www.msn.com