INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

An Introduction To The Parallel Virtual Machine

Zhang Jie

A Major Report
In
The Department
Of
Computer Science

Presented in Partial Fulfillment of the Requirement
for the Degree of Master of Computer Science at
Concordia University
Montreal,Quebec,Canada

July 1999

ABSTRACT

An Introduction To The Parallel Virtual Machine

Zhang Jie

This study is an introduction of the Parallel Algorithm and the software of the Parallel Virtual Machine. The emphasis is placed on how to use the Parallel Virtual Machine to solve some parallel algorithm problem. Some of the issues to be addressed are: the feature and functionality of the PVM, how to programming by using the PVM, how to install and setup the PVM..., also many parallel algorithm examples and small project are introduced. According to the PVM study, the software PVM is a very powerful tool. It can handle the parallel program running on different machines as in one machine. Also this study gives many programming algorithms to program a parallel software. Conclusions are parallel algorithm and parallel tools can make a lot of optimization on the computer's future.

Index

Part I Introduction

1. What is PVM?

This paper is intended to introduce what is PVM (Parallel Virtual Machine) and how to use PVM, which is a software package that allows a programmer to create and access a concurrent computing system made from networks of loosely coupled processing elements. This software package permits both homogeneous and heterogeneous collection of Unix computers and PCs hooked together by a network to be used as a single large parallel computer. The hardware collected into a user's virtual machine may be single processor workstations or PCs, vector machines or parallel supercomputers. The network connection can be as small as LAN or as large as WAN. Thus large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers. The software is very portable. The source, which is available free from netlib, has been compiled on everything from laptops to CRAYs. PVM enables users to exploit their existing computer hardware to solve much larger problems at minimal additional cost.

2. A brief history for the PVM

The PVM project began in the summer of 1989 at Oak Ridge National Laboratory. Vaidy Sunderam and Al Geist constructed the prototype system, PVM 1.0, this version of the system was used internally at the Lab and was not released to the outside. Version 2 of PVM was written at the University of Tennessee and released in March 1991. During the following year, PVM began to be used in many scientific applications. After user feedback and a number of changes (PVM 2.1 - 2.4), a complete rewrite was undertaken, and version 3 was completed in February 1993. It is PVM version 3.3. Now, what I introduced in this paper is PVM version 3.4 Beta 6 Release. The PVM software has been distributed freely and is being used in computational applications around the world. To successfully use the PVM, the user

should be some familiar with common programming techniques (C or Fortran) and understand some basic parallel processing concepts.

Part II PVM Installing

1. Application and Environments for PVM

PVM is ideally suited for concurrent application composed of many interrelated parts. PVM is particularly effective for heterogeneous applications that exploit specific strengths of individual machines on a network. As a loosely coupled concurrent supercomputer environment PVM is a viable scientific computing platform. PVM system used for number of applications such as molecular dynamics simulation, superconductivity studies, distributed fractal computations, matrix algorithms, and in the classroom as the basis for teaching concurrent computing. No special permission is required to create and use PVM, which can be built and installed on machines by anyone with a valid user id on those machines. The PVM has both the Unix version and PC version. So it can be installed both on the Unix systems and PCs.

2. How to get the software

There are several ways to get the PVM software package: ftp, WWW, xnetlib, or email. The latest version of the PVM source code and documentation is always available through *netlib*. Netlib is a software distribution service set up on the Internet that contains a wide range of computer software.

1. FTP: PVM files can be obtained by anonymous ftp to ftp.netlib.org. Look in directory /pvm3. The file index describes the files in this directory and its subdirectories.

2. WWW: Using a World Wide Web tool like Netscape or Internet Explore, the PVM files are accessed by using the address http://www.netlib.org/pvms/index.html Entering this site and download the files through the instructions. It is the easiest way to get the software package.

3. Xnetlib is a X-Window interface that allows a user to browse or query netlib for available software and to automatically transfer the selected software to the user's computer. To get xnetlib send email to netlib@netlib.org with the message send xnetlib.shar from xnetlib or anonymous ftp from ftp.netlib.org xnetlib/xnetlib.shar.

4. Also, the PVM software can be requested by email. To receive this software send email to netlib@netlib.org with the message: send index from pvm3. An automatic mail handler will return nine messages to you, each a portion of the encoded and compressed set of PVM source files. The advantage of this method is that anyone with email access to Internet can obtain the software.

The PVM software is distributed as a uuencoded, compressed, tar file for Unix. Also now the Install-shield version for win32 is available. After install it, the PVM documentation will includes a User's Guide, reference manual, and quick reference card.

3. How to install PVM on the Unix sytem

PVM's is simple to set up and use. That is one reason why the PVM is so popular today. PVM does not require special privileges to be installed. Anyone with a valid login on the hosts can do so. It can be both installed by the user in their $HOME directory and by the root in usr/local directory for all the user. After we get the PVM software, we should unzip it first. It maybe a .uu file or a .Z file. If it is a .uu format file, we should do such steps:

> uudecode pvm3.4.6.tar.z.uu

Whichever method used above, uncompress the resultant file by executing the command

> uncompress pvm3.4.6.tar.Z

then untar the file with

> tar -xvf pvm3.4.6.tar

Or it is a .Z format file. we should such steps:

> gunzip pvm3.4.6.tar.Z

Whilever we get a pvm3.4.6.tar file, executing the command

> tar –xvf pvm3.4.6.tar

After untar the file, it will create a pvm3 directory with source code and makefiles for many different machines included. These source files for PVM will take up about 7 Mbyte when uncompressed and unpacked.

PVM uses two environment variables when starting and running. Every user should 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.

In this paper, the installation is assumed to be under C Shell. The easiest method is to set these two variables in your .cshrc file. An example to set the PVM_ROOT in the .cshrc file is:

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.

After setting the two environment variables, we can build the workstation architecture. Building for each architecture type is done automatically by logging on to a host, going into the pvm3 directory, and typing 'make' command. The makefile will automatically determine which architecture it is being executed on, create

appropriate subdirectories, and build pvm, pvmd3, libpvm3.a, and libfpvm3.a, pvmgs, and libgpvm3.a. It places all these files in $PVM_ROOT/lib/PVM_ARCH, with the exception of pvmgs which is placed in $PVM_ROOT/bin/PVM_ARCH. After make the architecture, it will consume about 20M ~ 30M space.

4. How to install the PVM on the PCs

Because now the self-extracting executable available for Win95, NT is available, it is more easy to install the PVM on the PCs. InstallShield wizard does all the work while asking the user a series of questions about install options. We just need to put the required environment variables in the registry. And now it is easy to add our ex-friend's desktop. The Win32 PVM version fully interpolates with Unix version of PVM. Cluster can be a mix of Linux and NT hosts.

Unlike standardized compilers in the Unix world, different flags and libs are used for WIN32. Currently the PVM only distinguish between Borland 5.0 or VC++ 4.0 or higher. Thus, a dependency file in $(PVM_ROOT)/conf named WIN32.bat can be modified to point to the installed Compiler. With PVM 3.4 beta5 this file is updated and modified on the fly automatically. But when you add or change compilers, you either have to reinstall PVM using the installation wizard or modify the win32.bat file manually.

Former Environment variables like PVM_ROOT, PVM_ARCH, PVM_TMP, or PVM_RSH are no longer required to run PVM. The program now uses registry keys to locate required libraries or the installation directory. These registry keys have been set upon your installation information. Furthermore, a rsh client and a rexec client have been added to allow more control for adding additional resources.

PVM is built by invoking the console command "pvm" with an additional hostfile argument (you need to specify the full path e.g. d:\myfiles\pvmhostfile) which identifies potential resources. The machine on which "pvm" is run for the first time

(in a given session) is referred to as the "master daemon." If this machine crashes, your PVM virtual machine goes down (a single point of failure).

Manual adding of hosts is performed by the "add" command in the "pvm" console. Add commands must specify the location of the daemon, e.g.

pvm> add "hostname dx=d:\pvm3\lib\win32\pvmd3.exe"

for adding a win32 machine. You need to specify the location, including the executable. A Unix machine can simply be added by

pvm> add hostname

Now rshd or rexecd are still required for WinNT, rshd for WIN95. This is additional (shareware) software that can be find on the WWW, as long as MS does not offer its own solution. The better one that we recommend is: http://www.ataman.com. Please note that the rshd is required to run remote processes under your account. If you find pvm{d,l}.System in your PVM_TMP directory, then the rshd is NOT working properly.

Before using PVM, you should test the functionality of the additional software, e.g.

"rsh remotehost -l login dir | more"

This should show the directory listing of the remote host. Piping to "more" additionally checks for buffered std{io,err}.

Now the PVM are aware of Fortran problems using MS PowerFortran when calling one of the pvm_pk* functions. Since MS has discontinued Fortran we don't know if we will support it any longer.

Part III The PVM

1. PVM overview

PVM is a byproduct of ORNL's Heterogeneous Distributed Computing research project. With tens of thousands of users, PVM is the most popular software to combine networked computers. The PVM software, as a parallel programming tools, provides a unified framework within which parallel programs can be developed in an efficient and straightforward manner using existing hardware. Because PVM enables a collection of heterogeneous computer systems, so it provides the software environment that makes a cluster appear like a single large parallel computer. Also PVM continues to evolve based on changing computer and network technology and user feedback. PVM transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architectures.

The PVM computing model is simple yet very general, and accommodates a wide variety of application program structures. The programming interface is deliberately straightforward, thus permitting simple program structures to be implemented in an intuitive manner. The user writes his application as a collection of cooperating *tasks*. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronization between tasks. The PVM message-passing primitives are oriented towards heterogeneous operation, involving strongly typed constructs for buffering and transmission. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronization, and global sum.

PVM tasks may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, any task in existence may start or stop other tasks or add or delete computers from the virtual machine. Any process may communicate and/or synchronize with any other. Any specific control

and dependency structure may be implemented under the PVM system by appropriate use of PVM constructs and host language control-flow statements.

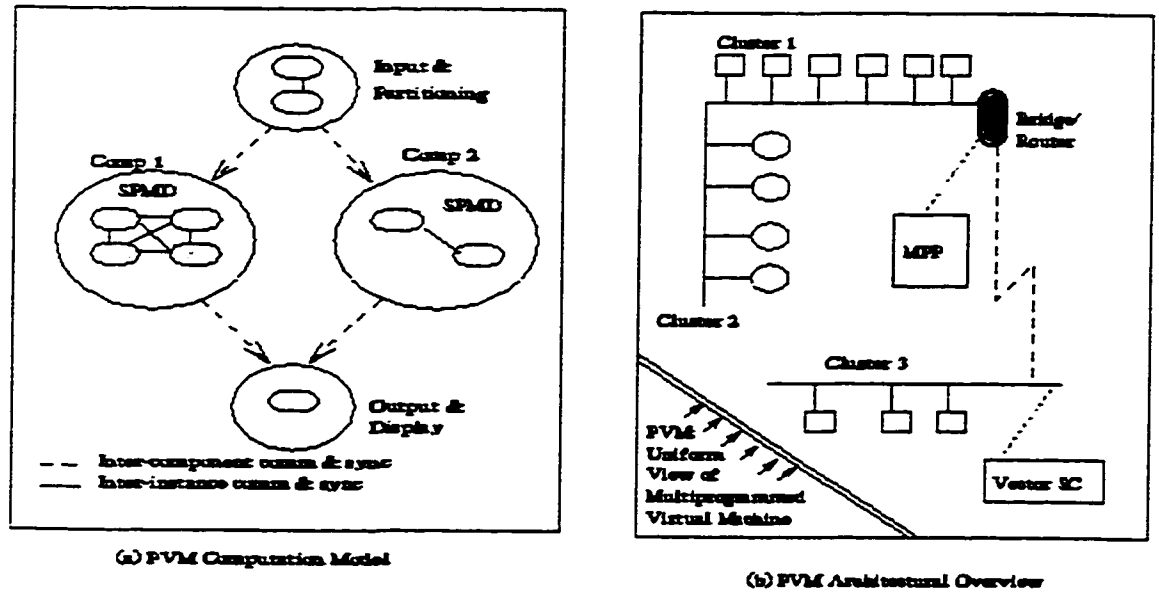
Owing to its ubiquitous nature (specifically, the virtual machine concept) and also because of its simple but complete programming interface, the PVM system has gained widespread acceptance in the high-performance scientific computing community.

2. PVM system

The PVM system is composed of two parts. The first is daemon and the second is a library of PVM interface routines. The daemon called *pvmd3* and sometimes abbreviated *pvmd*. It is include in the all computers that PVM package was installed. The example of a daemon program is the e-mail system-controlling program that 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 should start the PVM console on the host and create the virtual machine. Then the PVM application can 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 library of PVM interface routines 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 **Figure PVM system overview (a)** and an architectural view of the PVM system, highlighting the heterogeneity of the computing platforms supported by PVM, is shown in **Figure PVM system overview (b)**



(a) PVM Computation Model

(b) PVM Architectural Overview

Figure: 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.

3. PVM features

- **Popular PVM uses**

Hundreds of sites around the world are using PVM to solve important scientific, industrial, and medical problems in addition to PVM's use as an educational tool to teach parallel programming. With tens of thousands of users, PVM has become the de facto standard for distributed computing worldwide. Some popular PVM Uses is: 1. As Poor man's Supercomputer: it is idle cycles from network of workstations & PCs. 2. As Metacomputer linking multiple Supercomputers to ultimate computing performance 3. As Education Tool to teach parallel programming and academic research

- **Features supplied by PVM**

a) Portable – Runs on nearly every Unix machine, plus many shared – and distributed-memory multiprocessors

b) Heterogeneous Resource Management – Any types of machines can be combined in a single virtual machine, so the user can add/delete hosts from a virtual machine.

c) Scalable – Virtual machines can include hundreds of host computers, and run thousands of tasks.

d) Dynamic configuration and Process Control – computers can be dynamically added and deleted from the parallel virtual machine by the application or manually, also the user can spawn/kill tasks dynamically

e) Signals – PVM tasks can send signals to other tasks.

f) Multiple messages buffers – Allows easier development of PVM math libraries, graphical interface, etc.

g) Tracing – Call-level tracing built into PVM library.

h) Can be customized – User can write manager tasks to implement custom scheduling policies.

i) Message Passing - blocking send, blocking and non-blocking receive, collective

j) Dynamic Task Groups - task can join or leave a group at any time

k) Fault Tolerance - VM automatically detects faults and adjusts

- **The PVM new features and functions in Application capabilities on Ver. 3.4**

Now what we are using the PVM version 3.4.6. There are some new features supplied by the PVM new version. These features make the PVM more powerful and strength.

a) Communication Context –The communication context is a System-wide unique context tag. It can spawn tasks inherit context of parent and can let the existing PVM applications work be unchanged. It is now easy to add context to applications in this way:

```
new_context = pvm_newcontext( )        // It can broadcast the new
context to all tasks or put in the persistent message.
old_context = pvm_setcontext( new_context) // This make the safe
communication for user's application or library
```

info = pvm_freecontext(context)

context = pvm_getcontext( )safe messages

To this point, the messages in old context are buffered until their context is reset.

b) Message Handlers - for extending features. Message Handlers are make the user able to have their defined message handlers. The handles are triggered when message of matching (context, msgtag, source) arrives. There is no restrictions on the handler functions. Also, it allows users to define new control features inside a virtual machine. The message handle is shown as following:

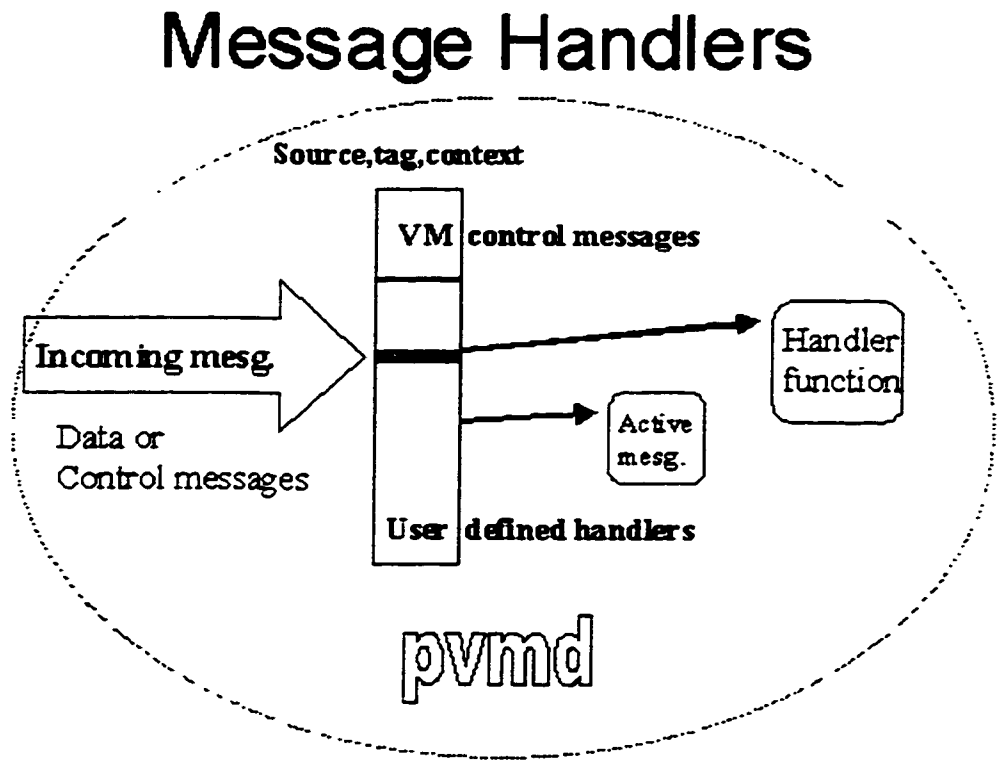


Figure Message handles

c) Persistent Messages – is a tuple space model. It let the tasks can store and retrieve messages by name. And it can distribut information database for dynamic programs. It provides rendezvous, attachment, groups, and many uses. Also multiple messages per "name" is possible within this feature. Like:

index = pvm_putinfo(name, msgbuf, flag)

pvm_recvinfo(name, index, flag)

pvm_delinfo(name, index, flag)

pvm_getmboxinfo(pattern, #names, array of struct).

These works can be shown as following **Figure Persistent Messages**:

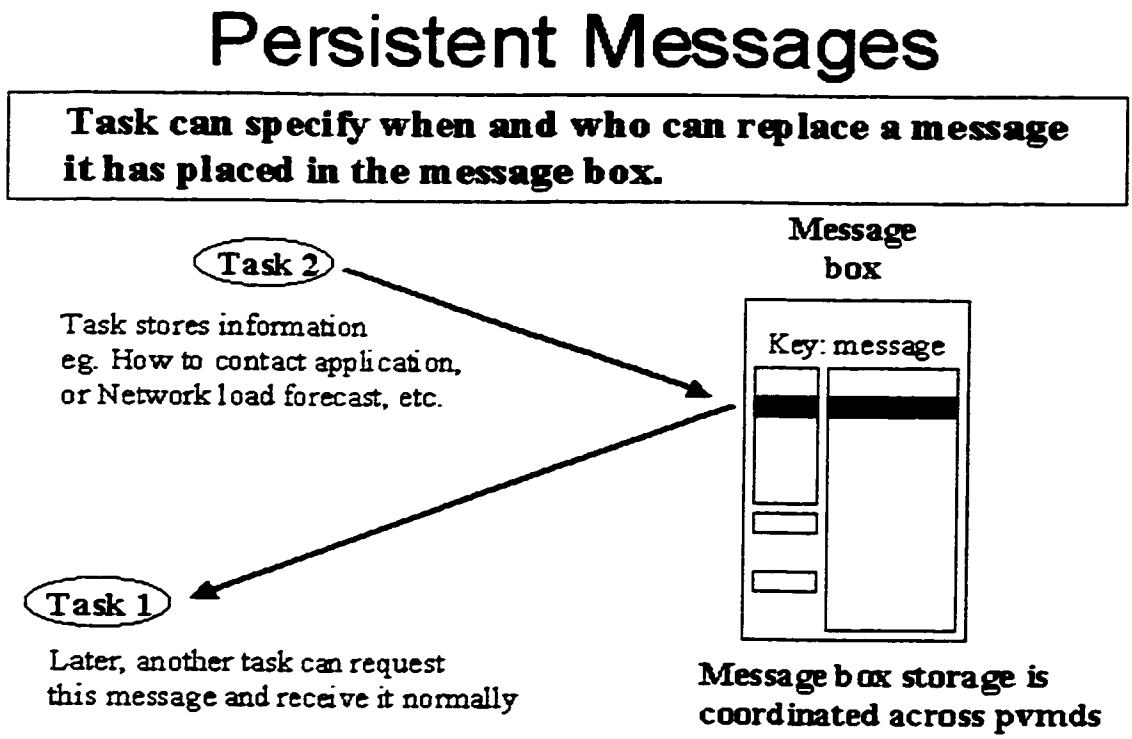


Figure Persistent Messages

d) User Defined Tracing - for specialized tools. PVM 3.4 uses standard SDDF trace format. So the tracing can be buffered to reduce intrusion and it can be

turned on/off within application. The user can select which events to trace and define their own trace events. eg. write a trace event everytime task computes "B".

f) Windows 95/NT Port – Now the PVM allow to work with WIN95, NT 3.5, and NT 4.0 very fast on multi-processor pentiums. And also it allows virtual machine to contain both UNIX and Windows hosts. It is great for PC clusters. The PVM InstallShield version makes adding PVM to a Windows host trivial.

- **PVM Interface**

In PVM 3 all PVM tasks are identified by an integer supplied by the local pvmd. 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 strings 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.

4. How PVM works

We describe 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, PVM avoids the use of operating system and language features that would be 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 interprocess communication and that each host in a virtual machine group can connect directly to every other host via TCP and UDP protocols. The requirement of full IP Figure How PVM is designed 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 (where the pvmd runs). The design and work process are shown as figure: How PVM is designed.

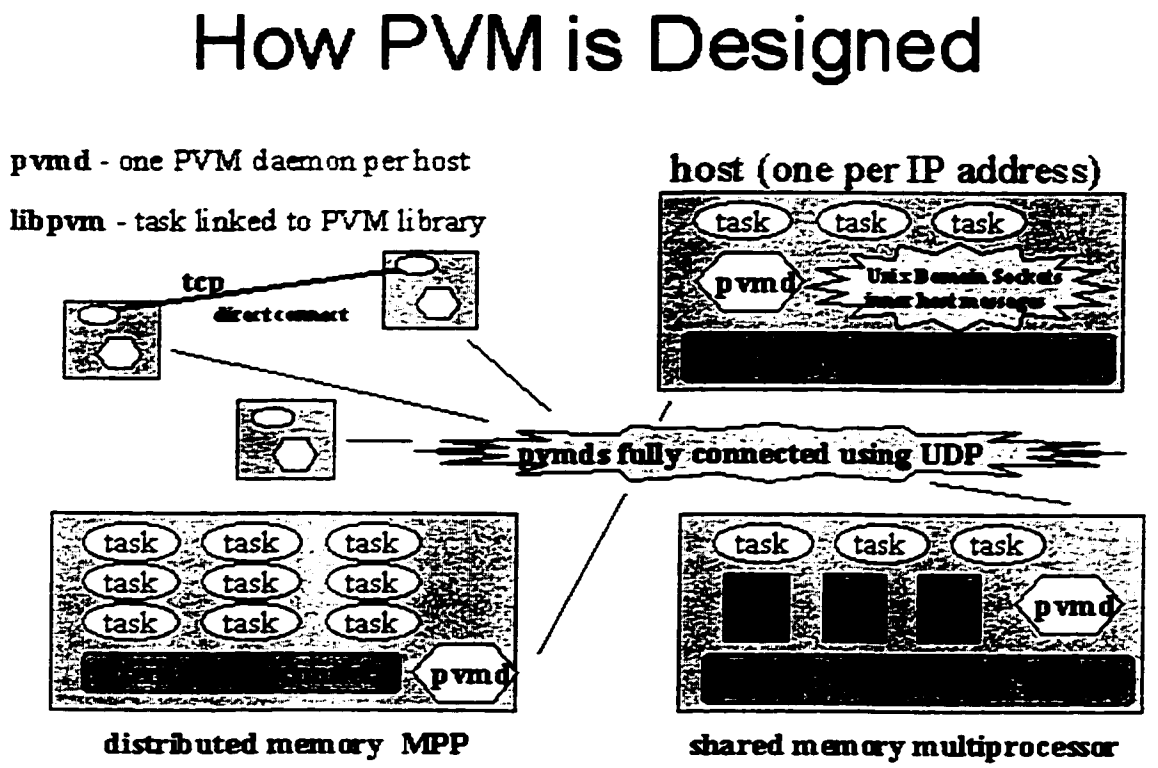


Figure How PVM is designed

5. Basic Programming Techniques

- **Write PVM Applications**

PVM Programming Model is a dynamic collection of serial and parallel computers appear as single distributed memory Virtual Machine. So all the tasks can be dynamically spawned and killed by any other task and any PVM task can send a message to any other. There is no limit to size or number of messages. The programming model supports fault tolerance, resource control, process control, heterogeneous networks and hosts. These sides are the most significant differences with basic programming techniques. The basic techniques are similar both for the logistical aspects of programming and for algorithm development, such as programming distributed-memory multiprocessors such as the nCUBE or the Intel family of multiprocessors.

PVM applications can be written in either C or Fortran 77. To execute a program under PVM, the user adds calls to PVM library routines that spawn off tasks to other machines within the user's virtual machine and allow tasks to send and receive data. While PVM is implemented in C, Fortran applications call the library routines through an interface that is included with the PVM source. This requires an additional library be linked in at compilation, though. The default directory PVM looks into for executables is $HOME/pvm3/bin/PVM_ARCH.

During execution, before any other PVM function can be called, a task must first enroll into PVM. This assigns a unique task *id* number to the task. These task *ids* are used when sending and receiving messages. Once a task has completed its work under PVM, it must inform the PVM daemon that it is exiting from the virtual machine. This does not terminate the process executing the task which can continue running, however, it may not interact with any other PVM tasks.

There are many examples are included in the PVM package that are illustrate the PVM usage . These examples sever as template to build the user's own application. These examples include:

hello hello_other	PVM equivalent to hello world
master slave	Master/slave example
spmd	SPMD example
gexample	Group and collective ops.
timing timing_slave	Timing example - comm perf
hitc hitc_slave	Dynamic load balance
inheritb	Communication context
imbi gmbi	Persistent messages template
mhf_server mhf_tickle	Message handlers

These can be found in the pvm3/examples directory.

- **C Examples**

1. Master – Slave Example

This example is from the pvm software package. At the first line of both programs includes the PVM header file. This file gives definitions to PVM symbolic names and functions. If PVM is available in a system directory, this header file will be installed as well. Consult with the system administrator for the proper directory.

The first PVM call in the master1 program informs the PVM daemon of its existence by enrolling the task in the virtual machine. The function pvm_mytid() is used for this purpose and assigns a task *id* to the calling task.

mytid = pvm_mytid()

The result returned is the assigned task *id*. There is no parameter to this function.

After the program is enrolled in the virtual machine, the master program initializes the data that is to be summed up. Next, the worker processes are spawned. The function to spawn worker processes is *pvm_spawn()*.

*int numt = pvm_spawn(char *task, char **argv, int flag,*
*char *where, int ntask, int *tids)*

The first parameter is a string containing the name of the executable file that is to be used. Any arguments that must be sent to this program are in an array pointed to by argv. If no arguments are required by the task, then the argv parameter is NULL. The flag parameter is used to determine the specific machine or type of architecture the spawned task is to be run on. Possible values for flag are:

- **PvmTaskDefault** - PVM chooses where this task is spawned to.
- **PvmTaskHost** - the where string argument specifies the particular machine.
- **PvmTaskArch** - the where string indicates the architecture type.

These symbolic names are defined in the PVM include file pvm3.h.

The fifth parameter, nproc, specifies the number of copies of the task to be spawned and tids is a pointer to an integer array that returns the task *ids* of all tasks spawned. The function returns the number of tasks that were successfully created. If some tasks could not be started, we can add two parameters (*ntask - numt*) of tids at the last positions which will contain error codes for the unsuccessful tasks.

In the example the master1 program will spawn code five (NPROCS) copies of the executable file worker. No arguments are to be sent and we are allowing PVM to choose which machines will be used to execute the worker code. The task *ids* will be placed in the array task_ids.

To send a message from one task to another, a send buffer is created to hold the data. The function pvm_initsend() creates and clears the buffer and returns a buffer identifier.

int bufid = pvm_initsend(int encoding)

If a single buffer is used, pvm_initsend() must be called each time a new message is to be sent, otherwise the new message will be appended to the message already in the send buffer.

The encoding parameter can be either **PvmDataDefault** or **PvmDataRaw**. The former option will use XDR encoding of message data if the virtual machine configuration is determined to be heterogeneous, else no encoding is done. The latter option does no encoding of the message data.

Before issuing a send command, the buffer must be packed with data to be sent. The functions to pack data into the active send buffer are pvm_pkint() where int indicates the type of data being packed. The data types supported by PVM (and their int function designation) are byte (byte), complex (cplx), double complex (dcplx), double (double), float (float), integer (int), long (long) and short(short). The example code packs integers and uses the function pvm_pkint(&nproc, 1, 1);pvm_pkint(tids, nproc, 1);

pvm_pkint(&n, 1, 1); Each of the functions outlined above has three parameters. The first is a pointer to the first item to be packed into the message, the second is the total number of items to be packed and the third is the stride to use when packing. There is also a function to pack a NULL terminated string (str) that requires only a single parameter which points to the first position of the string.

In the example code, within a loop, the master program clears the send buffer for each new message and packs this buffer with two things: 1) the number of array elements that follow in the message and 2) the array portion to be summed. Since each consecutive item from the array a is to be sent, starting with the num_data*i position, the stride for the packing function is 1. The task_ids array that was returned from the pvm_spawn() call is used to address each different task that will receive a portion of the array. The arbitrarily chosen value `4' is the msgtag used to label the messages.

After the array portions have been distributed, the master program must receive a partial sum from each of the worker processes. To receive a message, a task calls the pvm_recv() function.

int bufid = pvm_recv(int tid, int msgtag)

This will receive a message from task tid with label msgtag and place it into the receive buffer with *id* bufid. If no message is waiting from the given task with the expected label, the function waits until a message from the proper task and the correct label arrives. Values of '-1' for the parameters will match with any task *id* and/or label. In the example code, the master program is expecting a label value of '7' on messages from the worker tasks. The messages are to be received from tasks in the order that the send messages were issued.

Once a message has been received the data within must be unpacked. The unpacking functions are pvm_upkint() where the int corresponds to the type of data that is to be unpacked. The same int extensions used in the pvm_pkint() packing functions are valid for pvm_upkint(). For example, since the master program is receiving an integer from its worker processes, it calls the integer unpacking function

*int info = pvm_upkint(int *np, int nitem, int stride)*

The first parameter is a pointer to where the first item unpacked is to be stored. The second and third give the number of items to be unpacked and the stride to be used. Our example code unpacks each partial result received into a different element of the results array and adds it to the running sum.

After the sum is computed and printed the master task informs the PVM daemon that it is withdrawing from the virtual machine. This is done by executing the function

int info = pvm_exit()

As for the slave1 program, after enrolling in the virtual machine, the worker tasks wait to receive their portion of the array to be summed. Using the `-1' values in the pvm_recv() call indicates that the task does not care what task the message was sent from nor what label was used. Since the size of the array being sent may not be known ahead of time, after unpacking the number of data items from the message, the worker code allocates enough memory to hold the rest of the data contained in the message. The array fragment is summed up and the total is sent back to the parent. The task *id* of the task that spawned the current task is returned by the pvm_parent() function.

int parent_id = pvm_parent()

Note that since the master program is expecting a msgtag of `7' from the slave tasks, this value must be used in the pvm_send() call.

Helpful Hint: To run the Master-Slave C example under the Unix system can both run under the shell environment and PVM console.

Source files:

master1.c slave1.c

To compile:

% aimk master1 slave1

To run from shell (C version):

% master1

To Run from PVM console:

pvm> spawn -> master1

Sample output:

Spawning 3 worker tasks ... SUCCESSFUL

I got 100.000000 from 1; (expecting 100.000000)

I got 200.000000 from 0; (expecting 200.000000)

I got 300.000000 from 2; (expecting 300.000000)

Notes:

If you desire to run the master examples from the shell, then you need to compile the "master1h"

```
% aimk master1h
% master1h
```

Complete details for all PVM library functions can be found in Appenbix B part.

Figure : Master1.C

```
#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "slave1"

main()
{
   int mytid;              /* my task id */
        int tids[32];                          /* slave task ids */
   int n, nproc, numt, i, who, msgtype, nhost, narch;
   float data[100], result[32];
   struct pvmhostinfo *hostp;

   /* enroll in pvm */
   mytid = pvm_mytid();

   /* Set number of slaves to start */
   pvm_config( &nhost, &narch, &hostp );
```

```
nproc = nhost * 3;
if( nproc > 32 ) nproc = 32 ;
      printf("Spawning %d worker tasks ... " , nproc);

/* start up slave tasks */
numt=pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);
if( numt < nproc ){
  printf("\n Trouble spawning slaves. Aborting. Error codes are:\n");
  for( i=numt ; i<nproc ; i++ ) {
    printf("TID %d %d\n",i,tids[i]);
  }
  for( i=0 ; i<numt ; i++ ){
    pvm_kill( tids[i] );
  }
  pvm_exit();
  exit(1);
}
      printf("SUCCESSFUL\n");


/* Begin User Program */
n = 100;
/* initialize_data( data, n ); */
for( i=0 ; i<n ; i++ ){
  data[i] = 1.0;
}

/* Broadcast initial data to slave tasks */
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&nproc, 1, 1);
```

```
        pvm_pkint(tids, nproc, 1);
        pvm_pkint(&n, 1, 1);
        pvm_pkfloat(data, n, 1);
    pvm_mcast(tids, nproc, 0);

    /* Wait for results from slaves */
    msgtype = 5;
    for( i=0 ; i<nproc ; i++ ){
      pvm_recv( -1, msgtype );
      pvm_upkint( &who, 1, 1 );
      pvm_upkfloat( &result[who], 1, 1 );
      printf("I got %f from %d; ",result[who],who);
      if (who == 0)
         printf( "(expecting %f)\n", (nproc - 1) * 100.0);
      else
         printf( "(expecting %f)\n", (2 * who - 1) * 100.0);

    }
    /* Program Finished exit PVM before stopping */
    pvm_exit();}
```

Figure: Slave1.C

```
#include <stdio.h>
#include "pvm3.h"

main()
{
  int mytid;      /* my task id */
```

```
int tids[32];   /* task ids  */
      int n, me, i, nproc, master, msgtype;
      float data[100], result;
float work();

/* enroll in pvm */
      mytid = pvm_mytid();

/* Receive data from master */
msgtype = 0;
pvm_recv( -1, msgtype );
      pvm_upkint(&nproc, 1, 1);
      pvm_upkint(tids, nproc, 1);
      pvm_upkint(&n, 1, 1);
      pvm_upkfloat(data, n, 1);

/* Determine which slave I am (0 -- nproc-1) */
for( i=0; i<nproc ; i++ )
  if( mytid == tids[i] ){ me = i; break; }

/* Do calculations with data */
result = work( me, n, data, tids, nproc );

/* Send result to master */
pvm_initsend( PvmDataDefault );
pvm_pkint( &me, 1, 1 );
pvm_pkfloat( &result, 1, 1 );
msgtype = 5;
master = pvm_parent();
pvm_send( master, msgtype );
```

```
    /* Program finished. Exit PVM before stopping */
    pvm_exit();
}

float
work(me, n, data, tids, nproc )
    /* Simple example: slaves exchange data with left neighbor (wrapping) */
    int me, n, *tids, nproc;
    float *data;
{
    int i, dest;
    float psum = 0.0;
    float sum = 0.0;
    for( i=0 ; i<n ; i++ ){
      sum += me * data[i];
    }
    /* illustrate node-to-node communication */
    pvm_initsend( PvmDataDefault );
    pvm_pkfloat( &sum, 1, 1 );
    dest = me+1;
    if( dest == nproc ) dest = 0;
    pvm_send( tids[dest], 22 );
    pvm_recv( -1, 22 );
    pvm_upkfloat( &psum, 1, 1 );

    return( sum+psum );
}
```

2. Fork-Join

The first example demonstrates how to spawn off PVM tasks and synchronize with them. The program spawns several tasks, three by default. The children then synchronize by sending a message to their parent task. The parent receives a message from each of the spawned tasks and prints out information about the message from the child tasks.

The fork-join program contains the code for both the parent and the child tasks. Let's examine it in more detail. The very first thing the program does is call pvm_mytid(). This function must be called before any other PVM call can be made. The result of the pvm_mytid() call should always be a positive integer. If it is not, then something is seriously wrong. In fork-join we check the value of mytid; if it indicates an error, we call pvm_perror() and exit the program. The pvm_perror() call will print a message indicating what went wrong with the last PVM call. In our example the last call was pvm_mytid(), so pvm_perror() might print a message indicating that PVM hasn't been started on this machine. The argument to pvm_perror() is a string that will be pretended to any error message printed by pvm_perror(). In this case we pass argv[0], which is the name of the program as it was typed on the command line. The pvm_perror() function is modeled after the Unix perror() function.

Assuming we obtained a valid result for mytid, we now call pvm_parent(). The pvm_parent() function will return the TID of the task that spawned the calling task. Since we run the initial fork-join program from the Unix shell, this initial task will not have a parent; it will not have been spawned by some other PVM task but will have been started manually by the user. For the initial forkjoin task the result of pvm_parent() will not be any particular task id but an error code, PvmNoParent. Thus we can distinguish the parent forkjoin task from the children by checking whether the result of the pvm_parent() call is equal to PvmNoParent. If this task is the parent, then

it must spawn the children. If it is not the parent, then it must send a message to the parent.

Let's examine the code executed by the parent task. The number of tasks is taken from the command line as argv[1]. If the number of tasks is not legal, then we exit the program, calling pvm_exit() and then returning. The call to pvm_exit() is important because it tells PVM this program will no longer be using any of the PVM facilities. (In this case the task exits and PVM will deduce that the dead task no longer needs its services. Regardless, it is good style to exit cleanly.) Assuming the number of tasks is valid, forkjoin will then attempt to spawn the children.

The pvm_spawn() call tells PVM to start ntask tasks named argv[0]. The second parameter is the argument list given to the spawned tasks. In this case we don't care to give the children any particular command line arguments, so this value is null. The third parameter to spawn, PvmTaskDefault, is a flag telling PVM to spawn the tasks in the default location. Had we been interested in placing the children on a specific machine or a machine of a particular architecture, then we would have used PvmTaskHost or PvmTaskArch for this flag and specified the host or architecture as the fourth parameter. Since we don't care where the tasks execute, we use PvmTaskDefault for the flag and null for the fourth parameter. Finally, ntask tells spawn how many tasks to start; the integer array child will hold the task ids of the newly spawned children. The return value of pvm_spawn() indicates how many tasks were successfully spawned. If info is not equal to ntask, then some error occurred during the spawn. In case of an error, the error code is placed in the task id array, child, instead of the actual task id. The fork-join program loops over this array and prints the task ids or any error codes. If no tasks were successfully spawned, then the program exits.

For each child task, the parent receives a message and prints out information about that message. The pvm_recv() call receives a message (with that JOINTAG) from any task. The return value of pvm_recv() is an integer indicating a message buffer. This integer can be used to find out information about message buffers. The subsequent

call to pvm_bufinfo() does just this; it gets the length, tag, and task id of the sending process for the message indicated by buf. In fork-join the messages sent by the children contain a single integer value, the task id of the child task. The pvm_upkint() call unpacks the integer from the message into the mydata variable. As a sanity check, forkjoin tests the value of mydata and the task id returned by pvm_bufinfo(). If the values differ, the program has a bug, and an error message is printed. Finally, the information about the message is printed, and the parent program exits.

The last segment of code in forkjoin will be executed by the child tasks. Before placing data in a message buffer, the buffer must be initialized by calling pvm_initsend(). The parameter PvmDataDefault indicates that PVM should do whatever data conversion is needed to ensure that the data arrives in the correct format on the destination processor. In some cases this may result in unnecessary data conversions. If the user is sure no data conversion will be needed since the destination machine uses the same data format, then he can use PvmDataRaw as a parameter to pvm_initsend(). The pvm_pkint() call places a single integer, mytid, into the message buffer. It is important to make sure the corresponding unpack call exactly matches the pack call. Packing an integer and unpacking it as a float will not work correctly. Similarly, if the user packs two integers with a single call, he cannot unpack those integers by calling pvm_upkint() twice, once for each integer. There must be a one to one correspondence between pack and unpack calls. Finally, the message is sent to the parent task using a message tag of JOINTAG.

Fork Join Example

```
/*
Fork Join Example
Demonstrates how to spawn processes and exchange messages
*/

/* defines and prototypes for the PVM library */
#include <pvm3.h>
```

```
/* Maximum number of children this program will spawn */
#define MAXNCHILD   20
/* Tag to use for the joing message */
#define JOINTAG     11
int
main(int argc, char* argv[])
{
   /* number of tasks to spawn, use 3 as the default */
   int ntask = 3;
   /* return code from pvm calls */
   int info;
   /* my task id */
   int mytid;
   /* my parents task id */
   int myparent;
   /* children task id array */
   int child[MAXNCHILD];
   int i, mydata, buf, len, tag, tid;
   /* find out my task id number */
   mytid = pvm_mytid();

   /* check for error */
   if (mytid < 0) {
      /* print out the error */
      pvm_perror(argv[0]);
      /* exit the program */
      return -1;
      }
  /* find my parent's task id number */
```

```
myparent = pvm_parent();

/* exit if there is some error other than PvmNoParent */
if ((myparent < 0) && (myparent != PvmNoParent)) {
  pvm_perror(argv[0]);
  pvm_exit();
  return -1;
  }
/* if i don't have a parent then i am the parent */
if (myparent == PvmNoParent) {
  /* find out how many tasks to spawn */
  if (argc == 2) ntask = atoi(argv[1]);
  /* make sure ntask is legal */
       if ((ntask < 1) || (ntask > MAXNCHILD)) {
    pvm_exit(); return 0; }

  /* spawn the child tasks */
       info= pvm_spawn(argv[0], (char**)0, PvmTaskDefault,      (char*)0,
    ntask, child);
  /* print out the task ids */
  for (i = 0; i < ntask; i++)
   if (child[i] < 0)
                              /* print the error code in decimal*/
       printf(" %d", child[i]);
    else  /* print the task id in hex */
       printf("t%x\t", child[i]);
  putchar('\n');
  /* make sure spawn succeeded */
  if (info == 0) { pvm_exit(); return -1; }
```

```
/* only expect responses from those spawned correctly */
    ntask = info;

    for (i = 0; i < ntask; i++) {
      /* recv a message from any child process */
      buf = pvm_recv(-1, JOINTAG);
      if (buf < 0) pvm_perror("calling recv");
      info = pvm_bufinfo(buf, &len, &tag, &tid);
      if (info < 0)
              pvm_perror("calling pvm_bufinfo");
      info = pvm_upkint(&mydata, 1, 1);
      if (info < 0) pvm_perror("calling pvm_upkint");
      if (mydata != tid)
              printf("This should not happen!\n");
    printf("Length %d, Tag %d, Tid t%x\n", len, tag,
       tid);
      }
    pvm_exit();
    return 0;
    }
/* i'm a child */
info = pvm_initsend(PvmDataDefault);
if (info < 0) {
  pvm_perror("calling pvm_initsend"); pvm_exit(); return -1;
  }
info = pvm_pkint(&mytid, 1, 1);
if (info < 0) {
  pvm_perror("calling pvm_pkint"); pvm_exit();
  return -1;
  }
```

```
    info = pvm_send(myparent, JOINTAG);
    if (info < 0) {
      pvm_perror("calling pvm_send"); pvm_exit(); return -1;
      }
    pvm_exit();
    return 0;
}
```

Following part shows the output of running forkjoin. Notice that the order the messages were received is non-deterministic. Since the main loop of the parent processes messages on a first-come first-serve basis, the order of the prints are simply determined by time it takes messages to travel from the child tasks to the parent.

```
% forkjoin
t10001c t40149  tc0037
Length 4, Tag 11, Tid t40149
Length 4, Tag 11, Tid tc0037
Length 4, Tag 11, Tid t10001c
% forkjoin 4
t10001e t10001d t4014b  tc0038
Length 4, Tag 11, Tid t4014b
Length 4, Tag 11, Tid tc0038
Length 4, Tag 11, Tid t10001d
Length 4, Tag 11, Tid t10001e
```

Figure: Output of fork-join program

- **Running PVM Applications**

1) Include header file: PVM applications written in C should include header file pvm3.h, as follows: #include <pvm3.h>. It may need to specify the PVM include directory in the compiler flags as follows:
cc ... -I$PVM_ROOT/include ...

A header file for Fortran (fpvm3.h) is also supplied. You can include it using a statement like: INCLUDE 'fpvm3.h'. You may want to make a copy of fpvm3.h in your source directory. A header file for Fortran (fpvm3.h) is also supplied. Syntax for including files in Fortran is variable; the header file may need to be pasted into your source. A statement commonly used is:
INCLUDE '/usr/local/pvm/include/fpvm3.h'

2) Link: PVM applications written in C must be linked with at least the base PVM library, libpvm3.a. Fortran applications must be linked with both libfpvm3.a and libpvm3.a. Programs that use group functions must also be linked with libgpvm3.a. On some operating systems, PVM programs must be linked with still other libraries (for the socket or XDR functions).

Note that the order of libraries in the link command may be important. You may also need to specify the PVM library directory in the link command. A correct order is shown below (your compiler may be called something other than cc or f77).

cc/f77 [compiler flags] [source files] [loader flags]
-L$PVM_ROOT/lib/$PVM_ARCH -lfpvm3 -lgpvm3 -lpvm3
[libraries needed by PVM] [other libraries]

aimk automatically sets environment variable PVM_ARCH to the PVM architecture name and ARCHLIB to the necessary system libraries. You can use these variables to write a portable, shared makefile, called Makefile.aimk.

3) debug and tracing - PVM tasks can be started in a debugger on systems that support X-Windows. If PvmTaskDebug is specified in pvm_spawn(), PVM runs $PVM_ROOT/lib/debugger, which opens an *xterm* in which it runs the task in a debugger defined in pvm3/lib/debugger2. The PvmTaskDebug flag is not inherited, so you must modify each call to spawn. The DISPLAY environment variable can be exported to a remote host so the xterm will always be displayed on the local screen. Use the following command before running the application:
setenv PVM_EXPORT DISPLAY

Make sure DISPLAY is set to the name of your host (not unix:0) and the host name is fully qualified if your virtual machine includes hosts at more than one administrative site. To spawn a task in a debugger from the console, use the command:
spawn -? [rest of spawn command]

The console can spawn a task with tracing enabled (using the spawn -@), collect the trace data and print it out. In this way, a whole *job* (group of tasks related by parentage) can be traced. The console has a trace command to edit the mask passed to tasks it spawns. Or, XPVM can be used to collect and display trace data graphically.

It is difficult to start an application by hand and trace it, though. Tasks with no parent (anonymous tasks) have a default trace mask and sink of NULL. Not only must the first task call pvm_setopt() and pvm_settmask() to initialize the tracing parameters, but it must collect and interpret the trace data. If you must

start a traced application from a TTY, we suggest spawning an xterm from the console:

spawn -@ /usr/local/X11R5/bin/xterm -n PVMTASK

The task context held open by the xterm has tracing enabled. If you now run a PVM program in the xterm, it will reconnect to the task context and trace data will be sent back to the PVM console. Once the PVM program exits, you must spawn a new xterm to run again, since the task context will be closed.

Part IV. XPVM and JavaPVM

1. What is XPVM

ORNL is developing a software tool called XPVM. XPVM is a graphical tool to make the running and tuning of PVM programs easier. It provides a graphical interface to the PVM console commands and information, along with several animated views to monitor the execution of PVM programs. Scientists with little knowledge of PVM can click a few buttons to combine computers into a virtual machine and a few more buttons to launch their applications. These views provide information about the interactions among tasks in a parallel PVM program, to assist in debugging and performance tuning.

XPVM provides real-time animations of network and host loads as well as ParaGraph-like views to aid debugging and performance tuning. The first major update to XPVM is scheduled for FY95 improving its speed and functionality.

To analyze a program using XPVM, a user need only compile their program using the PVM library, version 3.3 or later (which has been instrumented to capture tracing information at run-time). Then, any task spawned from XPVM will return trace event information, for analysis in real time, or for post-mortem playback from saved trace files.

2. What is JavaPVM?

JavaPVM is an interface written using the Java® native methods capability, which allows Java applications to use the PVM software developed at Oak Ridge National Laboratory.

PVM is a software package that supports programs written in C, C++, and Fortran. JavaPVM extends the capabilities of PVM to the new, exciting, hype-filled world of Java. SunSoft's architecture-independent programming language for the Internet. JavaPVM allows Java applications (and possibly applets, though we haven't tested it) and existing C, C++, and Fortran applications to communicate with one another using the PVM API.

This means you could build Java interfaces to existing C, C++, and Fortran programs and use PVM to ship data from those programs to the Java interface. Or you could use this as a communications package as you transition applications from C or C++ to Java. Yeah, you could use the Java socket library, but PVM, and hence JavaPVM, is a bit simpler, more robust, and a lot better documented!

JavaPVM is not an implementation of PVM written in Java. There used to be such a package called JPVM, written by Adam Ferrari at U.Virginia, but we haven't seen it lately.

3. Where to get these software?

These softwares, XPVM and JavaPVM, are also can be get free from the web site. Here I have two recommended sites to get the XPVM and JavaPVM.

The place to get the XPVM is:
http://www.netlib.org/pvm3/xpvm/index.html

The place to get the JavaPVm is:
http://www.isye.gatech.edu/chmsr/jPVM/

Part I Where to get more information

Recommended paper, book and web site

PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing

Reference:

PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing

Advanced Tutorial on PVM 3.4 at site:

Appendix A The PVM command table

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.

Appendix B More PVM Examples

- **Dot Product**

Here we show a simple Fortran program, PSDOT, for computing a dot product. The program computes the dot product of arrays, X and Y. First PSDOT calls PVMFMYTID() and PVMFPARENT(). The PVMFPARENT call will return PVMNOPARENT if the task wasn't spawned by another PVM task. If this is the case, then PSDOT is the master and must spawn the other worker copies of PSDOT. PSDOT then asks the user for the number of processes to use and the length of vectors to compute. Each spawned process will receive *n/nproc* elements of X and Y, where *n* is the length of the vectors and *nproc* is the number of processes being used in the computation. If *nproc* does not divide *n* evenly, then the master will compute the dot product on extra the elements. The subroutine SGENMAT randomly generates values for X and Y. PSDOT then spawns *nproc* - 1 copies of itself and sends each new task a part of the X and Y arrays. The message contains the length of the subarrays in the message and the subarrays themselves. After the master spawns the worker processes and sends out the subvectors, the master then computes the dot product on its portion of X and Y. The master process then receives the other local dot products from the worker processes. Notice that the PVMFRECV call uses a wildcard (-1) for the task id parameter. This indicates that a message from *any* task will satisfy the receive. Using the wildcard in this manner results in a race condition. In this case the race condition does not cause a problem since addition is commutative. In other words, it doesn't matter in which order we add the partial sums from the workers. Unless one is certain that the race will not have an adverse effect on the program, race conditions should be avoided.

Once the master receives all the local dot products and sums them into a global dot product, it then calculates the entire dot product locally. These two results are then subtracted, and the difference between the two values is printed. A small difference can be expected because of the variation in floating-point roundoff errors.

If the PSDOT program is a worker then it receives a message from the master process containing subarrays of X and Y. It calculates the dot product of these subarrays and sends the result back to the master process. In the interests of brevity we do not include the SGENMAT and SDOT subroutines.

Example program: PSDOT.F:

```
      PROGRAM PSDOT
*
* PSDOT performs a parallel inner (or dot) product, where
* the vectors X and Y start out on a master node, which
* then sets up the virtual machine, farms out the data and
* work, and sums up the local pieces to get a global inner product.
*
*    .. External Subroutines ..
* EXTERNAL PVMFMYTID, PVMFPARENT, PVMFSPAWN, PVMFEXIT,
* PVMFINITSEND EXTERNAL PVMFPACK, PVMFSEND, PVMFRECV,
* PVMFUNPACK, SGENMAT
*
*    .. External Functions ..
      INTEGER ISAMAX
      REAL SDOT
      EXTERNAL ISAMAX, SDOT
*
*    .. Intrinsic Functions ..
      INTRINSIC MOD
*
*    .. Parameters ..
      INTEGER MAXN
```

```
      PARAMETER ( MAXN = 8000 )
      INCLUDE 'fpvm3.h'
*
*     .. Scalars ..
      INTEGER N, LN, MYTID, NPROCS, IBUF, IERR
      INTEGER I, J, K
      REAL LDOT, GDOT
*
*     .. Arrays ..
      INTEGER TIDS(0:63)
      REAL X(MAXN), Y(MAXN)
*
*     Enroll in PVM and get my and the master process' task
*     ID number
      CALL PVMFMYTID( MYTID )
      CALL PVMFPARENT( TIDS(0) )
*
*  If I need to spawn other processes (I am master
*  process)
*
      IF ( TIDS(0) .EQ. PVMNOPARENT ) THEN
*
*        Get starting information
*
            WRITE(*,*) 'How many processes should participate (1-64)?'
         READ(*,*) NPROCS
         WRITE(*,2000) MAXN
         READ(*,*) N
         TIDS(0) = MYTID
         IF ( N .GT. MAXN ) THEN
```

```
      WRITE(*,*) 'N too large.  Increase parameter MAXN to run'//
     $           'this case.'
            STOP
          END IF
*
*  LN is the number of elements of the dot product to do
*  locally.  Everyone has the same number, with the master
*  getting any left over elements.  J stores the number of
*     elements rest of procs do.
*
          J = N / NPROCS
          LN = J + MOD(N, NPROCS)
          I = LN + 1
*
*     Randomly generate X and Y
*
          CALL SGENMAT( N, 1, X, N, MYTID, NPROCS, MAXN, J )
          CALL SGENMAT( N, 1, Y, N, I, N, LN, NPROCS )
*
*     Loop over all worker processes
*
          DO 10 K = 1, NPROCS-1
*
*    Spawn process and check for error
*
      CALL PVMFSPAWN( 'psdot', 0, 'anywhere', 1, TIDS(K), IERR )
          IF (IERR .NE. 1) THEN
           WRITE(*,*) 'ERROR, could not spawn process #',K,
     $          '.  Dying . . .'
            CALL PVMFEXIT( IERR )
```

```
          STOP
        END IF
*
*       Send out startup info
        CALL PVMFINITSEND( PVMDEFAULT, IBUF )
        CALL PVMFPACK( INTEGER4, J, 1, 1, IERR )
        CALL PVMFPACK( REAL4, X(I), J, 1, IERR )
        CALL PVMFPACK( REAL4, Y(I), J, 1, IERR )
        CALL PVMFSEND( TIDS(K), 0, IERR )
        I = I + J
   10   CONTINUE
*
*       Figure master's part of dot product
*
      GDOT = SDOT( LN, X, 1, Y, 1 )
*
*       Receive the local dot products, and
*       add to get the global dot product
*
      DO 20 K = 1, NPROCS-1
        CALL PVMFRECV( -1, 1, IBUF )
        CALL PVMFUNPACK( REAL4, LDOT, 1, 1, IERR )
        GDOT = GDOT + LDOT
   20   CONTINUE
*
*       Print out result
*
      WRITE(*,*) ' '
      WRITE(*,*) '<x,y> = ',GDOT
*
```

```
*     Do sequential dot product and subtract from
*     distributed dot product to get desired error estimate
*
      LDOT = SDOT( N, X, 1, Y, 1 )
                WRITE(*,*) '<x,y> : sequential dot product.  <x,y>^ : '//
     $          'distributed dot product.'
                WRITE(*,*) '| <x,y> - <x,y>^ | = ',ABS(GDOT - LDOT)
      WRITE(*,*) 'Run completed.'
*
* If I am a worker process (i.e. spawned by master process)
*
    ELSE
*
* Receive startup info
*
      CALL PVMFRECV( TIDS(0), 0, IBUF )
      CALL PVMFUNPACK( INTEGER4, LN, 1, 1, IERR )
      CALL PVMFUNPACK( REAL4, X, LN, 1, IERR )
      CALL PVMFUNPACK( REAL4, Y, LN, 1, IERR )
*
*     Figure local dot product and send it in to master
*
      LDOT = SDOT( LN, X, 1, Y, 1 )
      CALL PVMFINITSEND( PVMDEFAULT, IBUF )
      CALL PVMFPACK( REAL4, LDOT, 1, 1, IERR )
      CALL PVMFSEND( TIDS(0), 1, IERR )
    END IF
*
    CALL PVMFEXIT( 0 )
*
```

```
1000  FORMAT(I10,' Successfully spawned process #',I2,', TID =',I10)
2000  FORMAT('Enter the length of vectors to multiply (1 -',I7,'):')
      STOP
*
*     End program PSDOT
*
      END
```

- **Failure**

The failure example demonstrates how one can kill tasks and how one can find out when tasks exit or fail. For this example we spawn several tasks, just as we did in the previous examples. One of these unlucky tasks gets killed by the parent. Since we are interested in finding out when a task fails, we call pvm_notify() after spawning the tasks. The pvm_notify() call tells PVM to send the calling task a message when certain tasks exit. Here we are interested in all the children. Note that the task calling pvm_notify() will receive the notification, *not* the tasks given in the task id array. It wouldn't make much sense to send a notification message to a task that has exited. The notify call can also be used to notify a task when a new host has been added or deleted from the virtual machine. This might be useful if a program wants to dynamically adapt to the currently available machines.

After requesting notification, the parent task then kills one of the children; in this case, one of the middle children is killed. The call to pvm_kill() simply kills the task indicated by the task id parameter. After killing one of the spawned tasks, the parent waits on a pvm_recv(-1, TASKDIED) for the message notifying it the task has died. The task id of the task that has exited is stored as a single integer in the notify message. The process unpacks the dead task's id and prints it out. For good measure it also prints out the task id of the task it killed. These ids should be the same. The child tasks simply wait for about a minute and then quietly exit.

Example program: failure.c

```
/*
   Failure notification example
   Demonstrates how to tell when a task exits
*/

/* defines and prototypes for the PVM library */
```

```
#include <pvm3.h>

/* Maximum number of children this program will spawn */
#define MAXNCHILD   20
/* Tag to use for the task done message */
#define TASKDIED        11
int
main(int argc, char* argv[])
{

  /* number of tasks to spawn, use 3 as the default */
  int ntask = 3;
  /* return code from pvm calls */
  int info;
  /* my task id */
  int mytid;
  /* my parents task id */
  int myparent;
  /* children task id array */
  int child[MAXNCHILD];
  int i, deadtid;
  int tid;
  char *argv[5];
  /* find out my task id number */
  mytid = pvm_mytid();

  /* check for error */
  if (mytid < 0) {
    /* print out the error */
    pvm_perror(argv[0]);
```

```
    /* exit the program */
    return -1;
    }
  /* find my parent's task id number */
  myparent = pvm_parent();
 /* exit if there is some error other than PvmNoParent */
  if ((myparent < 0) && (myparent != PvmNoParent)) {
    pvm_perror(argv[0]);
    pvm_exit();
    return -1;
    }

 /* if i don't have a parent then i am the parent */
 if (myparent == PvmNoParent) {
    /* find out how many tasks to spawn */
    if (argc == 2) ntask = atoi(argv[1]);
    /* make sure ntask is legal */
    if ((ntask < 1) || (ntask > MAXNCHILD)) { pvm_exit(); return 0; }

    /* spawn the child tasks */
         info = pvm_spawn(argv[0], (char**)0, PvmTaskDebug, (char*)0, ntask,
         child);
    /* make sure spawn succeeded */
    if (info != ntask) { pvm_exit(); return -1; }

    /* print the tids */
                    for (i = 0; i < ntask; i++) printf("t%x\t",child[i]); putchar('\n');

    /* ask for notification when child exits */
         info = pvm_notify(PvmTaskExit, TASKDIED, ntask, child);
```

```
        if (info < 0) { pvm_perror("notify"); pvm_exit();
                return -1; }
        /* reap the middle child */
        info = pvm_kill(child[ntask/2]);
        if (info < 0) { pvm_perror("kill"); pvm_exit();
                        return -1; }

        /* wait for the notification */
        info = pvm_recv(-1, TASKDIED);
        if (info < 0) { pvm_perror("recv"); pvm_exit();
                        return -1; }
        info = pvm_upkint(&deadtid, 1, 1);
        if (info < 0) pvm_perror("calling pvm_upkint");

        /* should be the middle child */
        printf("Task t%x has exited.\n", deadtid);
                printf("Task t%x is middle child.\n",   child[ntask/2]);
        pvm_exit();
        return 0;
        }

    /* i'm a child */
    sleep(63);
    pvm_exit();
    return 0;
}
```

- **Matrix Multiply**

In our next example we program a matrix-multiply algorithm described by Fox et al. The `mmult` program can be found at the end of this section. The `mmult` program will calculate $C = AB$, where C, A, and B are all square matrices. For simplicity we assume that m x m tasks will be used to calculate the solution. Each task will calculate a subblock of the resulting matrix C. The block size and the value of m is given as a command line argument to the program. The matrices A and B are also stored as blocks distributed over the tasks. Before delving into the details of the program, let us first describe the algorithm at a high level.

Assume we have a grid of m x m tasks. Each task ($\mathbf{t}_{ij}$ where $0 <= i,j < m$) initially contains blocks $\mathbf{C}_{ij}$, $\mathbf{A}_{ij}$, and $\mathbf{B}_{ij}$. In the first step of the algorithm the tasks on the diagonal ($\mathbf{t}_{ij}$ where $i = j$) send their block to all the other tasks in row i. After the transmission of $\mathbf{A}_{ij}$, all tasks calculate $\mathbf{A}_{ij} * \mathbf{B}_{ij}$ and add the result into $\mathbf{C}_{ij}$. In the next step, the column blocks of B are rotated. That is, $\mathbf{t}_{ij}$ sends its block of B to $\mathbf{t}_{(i-1)j}$. (Task t_{0j} sends its B block to $\mathbf{t}_{(m-1)j}$. The tasks now return to the first step; $\mathbf{A}_{i(i+1)}$ is multicast to all other tasks in row i, and the algorithm continues. After m iterations the C matrix contains A x B, and the B matrix has been rotated back into place.

Let's now go over the matrix multiply as it is programmed in PVM. In PVM there is no restriction on which tasks may communicate with which other tasks. However, for this program we would like to think of the tasks as a two-dimensional conceptual torus. In order to enumerate the tasks, each task joins the group `mmult`. Group ids are used to map tasks to our torus. The first task to join a group is given the group id of zero. In the mmult program, the task with group id zero spawns the other tasks and

sends the parameters for the matrix multiply to those tasks. The parameters are *m* and *bklsize*: the square root of the number of blocks and the size of a block, respectively. After all the tasks have been spawned and the parameters transmitted, pvm_barrier() is called to make sure all the tasks have joined the group. If the barrier is not performed, later calls to pvm_gettid() might fail since a task may not have yet joined the group.

After the barrier, we store the task ids for the other tasks in our ``row" in the array myrow. This is done by calculating the group ids for all the tasks in the row and asking PVM for the task id for the corresponding group id. Next we allocate the blocks for the matrices using malloc(). In an actual application program we would expect that the matrices would already be allocated. Next the program calculates the row and column of the block of C it will be computing. This is based on the value of the group id. The group ids range from 0 to m - 1 inclusive. Thus the integer division of (*mygid*/m) will give the task's row and (*mygid* mod m) will give the column, if we assume a row major mapping of group ids to tasks. Using a similar mapping, we calculate the group id of the task directly *above* and *below* in the torus and store their task ids in up and down, respectively.

Next the blocks are initialized by calling InitBlock(). This function simply initializes A to random values, B to the identity matrix, and C to zeros. This will allow us to verify the computation at the end of the program by checking that $A = C$.

Finally we enter the main loop to calculate the matrix multiply. First the tasks on the diagonal multicast their block of A to the other tasks in their row. Note that the array myrow actually contains the task id of the task doing the multicast. Recall that pvm_mcast() will send to all the tasks in the tasks array except the calling task. This procedure works well in the case of mmult since we don't want to have to needlessly handle the extra message coming into the multicasting task with an extra pvm_recv(). Both the multicasting task and the tasks receiving the block calculate the AB for the diagonal block and the block of B residing in the task.

After the subblocks have been multiplied and added into the *C* block, we now shift the *B* blocks vertically. Specifically, we pack the block of *B* into a message, send it to the up task id, and then receive a new *B* block from the down task id.

Note that we use different message tags for sending the *A* blocks and the *B* blocks as well as for different iterations of the loop. We also fully specify the task ids when doing a pvm_recv(). It's tempting to use wildcards for the fields of pvm_recv(); however, such a practice can be dangerous. For instance, had we incorrectly calculated the value for up and used a wildcard for the pvm_recv() instead of down, we might have sent messages to the wrong tasks without knowing it. In this example we fully specify messages, thereby reducing the possibility of mistakes by receiving a message from the wrong task or the wrong phase of the algorithm.

Once the computation is complete, we check to see that $A = C$, just to verify that the matrix multiply correctly calculated the values of *C*. This check would not be done in a matrix multiply library routine, for example.

It is not necessary to call pvm_lvgroup(), since PVM will realize the task has exited and will remove it from the group. It is good form, however, to leave the group before calling pvm_exit(). The reset command from the PVM console will reset all the PVM groups. The pvm_gstat command will print the status of any groups that currently exist.

- **Example program: mmult.c**

```
/*
  Matrix Multiply
```

```
*/

/* defines and prototypes for the PVM library */
#include <pvm3.h>
#include <stdio.h>

/* Maximum number of children this program will spawn */
#define MAXNTIDS   100
#define MAXROW     10

/* Message tags */
#define ATAG       2
#define BTAG       3
#define DIMTAG     5
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[])
{

   /* number of tasks to spawn, use 3 as the 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;
```

```
    /* find out my task id number */
    mytid = pvm_mytid();
    pvm_advise(PvmRouteDirect);

    /* check for error */
    if (mytid < 0) {
      /* print out the error */
      pvm_perror(argv[0]);
      /* exit the program */
      return -1;
      }
    /* 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(stderr, "usage: mmult m blk\n");
        pvm_lvgroup("mmult"); pvm_exit(); return -1;
        }
```

```
    /* make sure ntask is legal */
    ntask = m*m;
    if ((ntask < 1) || (ntask >= MAXNTIDS)) {
     fprintf(stderr, "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 {
    /* recv the 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 valid pointers */
  if (!(a && b && c && atmp)) {
    fprintf(stderr, "%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 neighbor'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 columns 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);
   }
/* 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("Done.\n");
free(a); free(b); free(c); free(atmp);
```

```
    pvm_lvgroup("mmult");
    pvm_exit();
    return 0;
}
```

- **One-Dimensional Heat Equation**

Here we present a PVM program that calculates heat diffusion through a substrate, in this case a wire. Consider the one-dimensional heat equation on a thin wire:

$$\frac{\partial A}{\partial t} = \frac{\partial^2 A}{\partial x^2} \tag{6.5.1}$$

and a discretization of the form

$$\frac{A_{i+1,j} - A_{i,j}}{\Delta t} = \frac{A_{i,j+1} - 2A_{i,j} + A_{i,j-1}}{\Delta x^2} \tag{6.5.2}$$

giving the explicit formula

$$A_{i+1,j} = A_{i,j} + \frac{\Delta t}{\Delta x^2}(A_{i,j+1} - 2A_{i,j} + A_{i,j-1}). \tag{6.5.3}$$

initial and boundary conditions:

$A(t,0) = 0$, $A(t,1) = 0$ for all t
$A(0,x) = \sin(\pi x)$ for $0 \leq x \leq 1$.

The pseudo code for this computation is as follows:

```
for i = 1:tsteps-1;
    t = t+dt;
    a(i+1,1)=0; a(i+1,n+2)=0;
    for j = 2:n+1;
     a(i+1,j)=a(i,j) + mu*(a(i,j+1)-2*a(i,j)+a(i,j-1));
    end;
    t;
    a(i+1,1:n+2);
    plot(a(i,:))
end
```

For this example we will use a master-slave programming model. The master, heat.c, spawns five copies of the program heatslv. The slaves compute the heat diffusion for subsections of the wire in parallel. At each time step the slaves exchange boundary information, in this case the temperature of the wire at the boundaries between processors.

Let's take a closer look at the code. In heat.c the array `solution` will hold the solution for the heat diffusion equation at each time step. This array will be output at the end of the program in xgraph format. (xgraph is a program for plotting data.) First the heatslv tasks are spawned. Next, the initial data set is computed. Notice that the ends of the wires are given initial temperature values of zero.

The main part of the program is then executed four times, each with a different value for Δt. A timer is used to compute the elapsed time of each compute phase. The initial data sets are sent to the heatslv tasks. The left and right neighbor task ids are sent along with the initial data set. The heatslv tasks use these to communicate boundary information.

(Alternatively, we could have used the PVM group calls to map tasks to segments of the wire. By using the group calls we would have avoided explicitly communicating the task ids to the slave processes.)

After sending the initial data, the master process simply waits for results. When the results arrive, they are integrated into the solution matrix, the elapsed time is calculated, and the solution is written out to the xgraph file. Once the data for all four phases has been computed and stored, the master program prints out the elapsed times and kills the slave processes.

Example program: heat.c

```
/*
heat.c
Use PVM to solve a simple heat diffusion differential
equation, using 1 master program and 5 slaves.
```

```
The master program sets up the data, communicates it to the slaves and waits for the
results to be sent from the slaves. Produces xgraph ready files of the results.
*/

#include "pvm3.h"
#include <stdio.h>
#include <math.h>
#include <time.h>
#define SLAVENAME "heatslv"
#define NPROC 5
#define TIMESTEP 100
#define PLOTINC 10
#define SIZE 1000

int num_data = SIZE/NPROC;

main()
{  int mytid, task_ids[NPROC], i, j;
   int left, right, k, l;
   int step = TIMESTEP;
   int info;

   double init[SIZE], solution[TIMESTEP][SIZE];
   double result[TIMESTEP*SIZE/NPROC], deltax2;
   FILE *filenum;
   char *filename[4][7];
   double deltat[4];
   time_t t0;
   int etime[4];
```

```
    filename[0][0] = "graph1";
    filename[1][0] = "graph2";
    filename[2][0] = "graph3";
    filename[3][0] = "graph4";

    deltat[0] = 5.0e-1;
    deltat[1] = 5.0e-3;
    deltat[2] = 5.0e-6;
    deltat[3] = 5.0e-9;

/* enroll in pvm */
    mytid = pvm_mytid();

/* spawn the slave tasks */
 info = pvm_spawn(SLAVENAME,(char **)0, PvmTaskDefault, "",
       NPROC,task_ids);
/* create the initial data set */
   for (i = 0; i < SIZE; i++)
    init[i] = sin(M_PI * ( (double)i / (double)(SIZE-1) ));
   init[0] = 0.0;
   init[SIZE-1] = 0.0;

/*run the problem 4 times for different values of delta t*/
    for (l = 0; l < 4; l++) {
       deltax2 = (deltat[l]/pow(1.0/(double)SIZE,2.0));
       /* start timing for this run */
       time(&t0);
       etime[l] = t0;
/* send the initial data to the slaves. */
/* include neighbor info for exchanging boundary data */
```

```
        for (i = 0; i < NPROC; i++) {
          pvm_initsend(PvmDataDefault);
          left = (i == 0) ? 0 : task_ids[i-1];
          pvm_pkint(&left, 1, 1);
          right = (i == (NPROC-1)) ? 0 : task_ids[i+1];
          pvm_pkint(&right, 1, 1);
          pvm_pkint(&step, 1, 1);
          pvm_pkdouble(&deltax2, 1, 1);
          pvm_pkint(&num_data, 1, 1);
          pvm_pkdouble(&init[num_data*i], num_data, 1);
          pvm_send(task_ids[i], 4);
          }

/* wait for the results */
        for (i = 0; i < NPROC; i++) {
          pvm_recv(task_ids[i], 7);
          pvm_upkdouble(&result[0], num_data*TIMESTEP, 1);
/* update the solution */
          for (j = 0; j < TIMESTEP; j++)
            for (k = 0; k < num_data; k++)
              solution[j][num_data*i+k] = result[wh(j,k)];
          }

/* stop timing */
        time(&t0);
        etime[l] = t0 - etime[l];

/* produce the output */
              filenum = fopen(filename[l][0], "w");
        fprintf(filenum,"TitleText: Wire Heat over Delta
```

```
                Time: %e\n",
            deltat[l]);
        fprintf(filenum,"XUnitText: Distance\nYUnitText:
                        Heat\n");
        for (i = 0; i < TIMESTEP; i = i + PLOTINC) {
            fprintf(filenum,"\"Time index: %d\n",i);
            for (j = 0; j < SIZE; j++)
                fprintf(filenum,"%d %e\n",j,
                                    solution[i][j]);
            fprintf(filenum,"\n");
            }
        fclose (filenum);
    }

/* print the timing information */
    printf("Problem size: %d\n",SIZE);
    for (i = 0; i < 4; i++)
        printf("Time for run %d: %d sec\n",i,etime[i]);

/* kill the slave processes */
    for (i = 0; i < NPROC; i++) pvm_kill(task_ids[i]);
    pvm_exit();
}

int wh(x, y)
int x, y;
{
    return(x*num_data+y);
}
```

The heatslv programs do the actual computation of the heat diffusion through the wire. The slave program consists of an infinite loop that receives an initial data set, iteratively computes a solution based on this data set (exchanging boundary information with neighbors on each iteration), and sends the resulting partial solution back to the master process.

Rather than using an infinite loop in the slave tasks, we could send a special message to the slave ordering it to exit. To avoid complicating the message passing, however, we simply use the infinite loop in the slave tasks and kill them off from the master program. A third option would be to have the slaves execute only once, exiting after processing a single data set from the master. This would require placing the master's spawn call inside the main for loop of heat.c. While this option would work, it would needlessly add overhead to the overall computation.

For each time step and before each compute phase, the boundary values of the temperature matrix are exchanged. The left-hand boundary elements are first sent to the left neighbor task and received from the right neighbor task. Symmetrically, the right-hand boundary elements are sent to the right neighbor and then received from the left neighbor. The task ids for the neighbors are checked to make sure no attempt is made to send or receive messages to nonexistent tasks.

Example program: heatslv.c

```
/*
heatslv.c

   The slaves receive the initial data from the host,
   exchange boundary information with neighbors,
   and calculate the heat change in the wire.
   This is done for a number of iterations, sent by the
    master.*/
```

```
#include "pvm3.h"
#include <stdio.h>

int num_data;

main()
{
   int mytid, left, right, i, j, master;
   int timestep;

   double *init, *A;
   double leftdata, rightdata, delta, leftside, rightside;

/* enroll in pvm */
   mytid = pvm_mytid();
   master = pvm_parent();

/* receive my data from the master program */
  while(1) {
   pvm_recv(master, 4);
   pvm_upkint(&left, 1, 1);
   pvm_upkint(&right, 1, 1);
   pvm_upkint(&timestep, 1, 1);
   pvm_upkdouble(&delta, 1, 1);
   pvm_upkint(&num_data, 1, 1);
   init = (double *) malloc(num_data*sizeof(double));
   pvm_upkdouble(init, num_data, 1);

/* copy the initial data into my working array */
```

```
    A = (double *) malloc(num_data * timestep *
      sizeof(double));
    for (i = 0; i < num_data; i++) A[i] = init[i];

/* perform the calculation */

  for (i = 0; i < timestep-1; i++) {
    /* trade boundary info with my neighbors */
    /*  send left, receive right    */
    if (left != 0) {
       pvm_initsend(PvmDataDefault);
       pvm_pkdouble(&A[wh(i,0)],1,1);
       pvm_send(left, 5);
       }
    if (right != 0) {
       pvm_recv(right, 5);
       pvm_upkdouble(&rightdata, 1, 1);
    /* send right, receive left */
       pvm_initsend(PvmDataDefault);
       pvm_pkdouble(&A[wh(i,num_data-1)],1,1);
       pvm_send(right, 6);
       }
    if (left != 0) {
       pvm_recv(left, 6);
       pvm_upkdouble(&leftdata,1,1);
       }

/* do the calculations for this iteration */
    for (j = 0; j < num_data; j++) {
       leftside = (j == 0) ? leftdata : A[wh(i,j-1)];
```

```
        rightside = (j == (num_data-1)) ? rightdata :
                                        A[wh(i,j+1)];
        if ((j==0)&&(left==0))
          A[wh(i+1,j)] = 0.0;
        else if ((j==(num_data-1))&&(right==0))
          A[wh(i+1,j)] = 0.0;
        else
          A[wh(i+1,j)]=
            A[wh(i,j)]+delta*(rightside-
                                2*A[wh(i,j)]+leftside);
        }
    }

/* send the results back to the master program */

     pvm_initsend(PvmDataDefault);
     pvm_pkdouble(&A[0],num_data*timestep,1);
     pvm_send(master,7);
    }

/* just for good measure */
   pvm_exit();
}

int wh(x, y)
int x, y;
{
    return(x*num_data+y);
}
```

Appendix C My Examples

1. 2D Heat Equation DESCRIPTION:

This project is based on a simplified two-dimensional heat equation domain decomposition. The initial temperature is computed to be high in the middle of the domain and zero at the boundaries. The boundaries are held at zero throughout the simulation. During the time-stepping, an array containing two domains is used; these domains alternate between old data and new data.

1) pvm_heat2D.c

```
#include <stdio.h>
#include "pvm3.h"                    /* PVM version 3.0 include file */
extern void draw_heat(int, int);     /* X routine to create graph */

#define NXPROB      20               /* x dimension of problem grid */
#define NYPROB      20               /* y dimension of problem grid */
#define STEPS       50              /* number of time steps */
#define AOUT        "heat2D"         /* name of PVM executable */
#define DONTCARE    -1               /* accept message from anytask */
#define MAXCHILD    8                /* maximum number of children tasks */
#define MINCHILD    3                /* minimum number of children tasks */
#define BEGIN       1               /* message type */
#define NGHBOR1     2                /* message type */
#define NGHBOR2     3                /* message type */
#define NONE        0               /* indicates no neighbor */
#define DONE        4               /* message type */

struct Parms {
 float cx;
```

```
  float cy;
  } parms = {0.1, 0.1};

main() {
void inidat(), prtdat(), update();
float  u[2][NXPROB][NYPROB];        /* array for grid */
int    taskid,parent,tids[MAXCHILD], /* PVM taskids */
       nproc,narch,                 /* number of PVM machines & architect */
       nchild,                      /* number of children processes */
       averow,rows,offset,extra,    /* for sending rows of data */
       neighbor1,neighbor2,         /* neighbor tasks */
       rc,start,end,                /* misc */
       i,ix,iy,iz,it;               /* loop variables */
struct hostinfo {                   /* structure defined in pvm3.h */
       int hi_tid;                  /* -needed for call to pvm_config */
       char *hi_name;
       char *hi_arch;
       int hi_speed;
       } hostp;


/* First, enroll this process in PVM and find parent task id */
taskid = pvm_mytid();
parent = pvm_parent();

if (parent == PvmNoParent) {
  /* Determine the number of machines in this configuration. Then spawn one
     child process for each processor excluding the parent.  If number of
     children exceed MAXCHILD, then just spawn the maximum. If less than
     MINCHILD then just spawn MINCHILD child processes */
```

```
rc = pvm_config(&nproc,&narch,&hostp);
nchild = nproc-1;
if (nchild > MAXCHILD)
 nchild = MAXCHILD;
else if (nchild < MINCHILD)
 nchild = MINCHILD;
printf("Spawning %d children processes\n",nchild);
for (i=0; i<nchild; i++) {
 rc = pvm_spawn(AOUT,NULL,PvmTaskDefault,"",1,&tids[i]);
 printf("...child task id= %d\n",tids[i]);
 }

/* Initialize grid */
printf("Grid size: X= %d  Y= %d  Time steps= %d\n",NXPROB,NYPROB,STEPS);
printf("Initializing grid and writing initial.dat file...\n");
inidat(NXPROB, NYPROB, u);
prtdat(NXPROB, NYPROB, u, "initial.dat");

/* Distribute work to children.  Must first figure out how many rows to
/* send and what to do with extra rows.  */
averow = NXPROB/nchild;
extra = NXPROB%nchild;
offset = 0;
for (i=0; i<nchild; i++) {
 rows = (i < extra) ? averow+1 : averow;
 /* Tell each child which other children are its neighbors, since
 /* they must exchange data with each other. */
 if (i == 0)
  neighbor1 = NONE;
 else
```

```
    neighbor1 = tids[i-1];
   if (i == nchild-1)
    neighbor2 = NONE;
   else
    neighbor2 = tids[i+1];
/*  Now send startup information to each child  */
   rc = pvm_initsend(PvmDataRaw);
   rc = pvm_pkint(&offset, 1, 1);
   rc = pvm_pkint(&rows, 1, 1);
   rc = pvm_pkint(&neighbor1, 1, 1);
   rc = pvm_pkint(&neighbor2, 1, 1);
   rc = pvm_pkfloat(&u[0][offset][0],rows*NYPROB,1);
   rc = pvm_send(tids[i],BEGIN);
   printf("Sent to= %d offset= %d rows= %d neighbor1= %d neighbor2= %d\n",
        tids[i],offset,rows,neighbor1,neighbor2);
   offset = offset + rows;
   }

 /* Now wait for results from all children tasks */
 for (i=0; i<nchild; i++) {
  pvm_recv(DONTCARE,DONE);
  rc = pvm_upkint(&offset, 1, 1);
  rc = pvm_upkint(&rows, 1, 1);
  rc = pvm_upkfloat(&u[0][offset][0],rows*NYPROB,1);
  }

 /* Exit PVM */
 rc = pvm_exit();

 /* Write final output and call X graph */
```

```
 printf("Writing final.dat file and generating graph...\n");
 prtdat(NXPROB, NYPROB, &u[0][0][0], "final.dat");
 draw_heat(NXPROB,NYPROB);

 }  /* End of parent code */


if (parent != PvmNoParent) {
 /************************** child code *********************************/
 /* Initialize everything - including the borders - to zero */
 for (iz=0; iz<2; iz++)
  for (ix=0; ix<NXPROB; ix++)
   for (iy=0; iy<NYPROB; iy++)
    u[iz][ix][iy] = 0.0;

 /* Now receive my offset, rows, neighbors and grid partition from parent */
 rc = pvm_recv(parent,BEGIN);
 rc = pvm_upkint(&offset,1,1);
 rc = pvm_upkint(&rows,1,1);
 rc = pvm_upkint(&neighbor1, 1, 1);
 rc = pvm_upkint(&neighbor2, 1, 1);
 rc = pvm_upkfloat(&u[0][offset][0],rows*NYPROB,1);

 /* Determine border elements.  Need to consider first and last columns.
 /* Obviously, row 0 can't exchange with row 0-1.  Likewise, the last
 /* row can't exchange with last+1.  */
 if (offset==0)
  start=1;
 else
  start=offset;
```

```
if ((offset+rows)==NXPROB)
 end=start+rows-2;
else
 end = start+rows-1;

/* Begin doing STEPS iterations. Must communicate border rows with
   neighbors. If I have the first or last grid row, then I only need to
   communicate with one neighbor */
iz = 0;
for (it = 1; it <= STEPS; it++) {
 if (neighbor1 != NONE) {
   rc = pvm_initsend(PvmDataRaw);
   rc = pvm_pkfloat(&u[iz][offset][0],NYPROB,1);
   rc = pvm_send(neighbor1,NGHBOR2);

   rc = pvm_recv(neighbor1,NGHBOR1);
   rc = pvm_upkfloat(&u[iz][offset-1][0],NYPROB,1);
   }

 if (neighbor2 != NONE) {
   rc = pvm_initsend(PvmDataRaw);
   rc = pvm_pkfloat(&u[iz][offset+rows-1][0],NYPROB,1);
   rc = pvm_send(neighbor2,NGHBOR1);

   rc = pvm_recv(neighbor2,NGHBOR2);
   rc = pvm_upkfloat(&u[iz][offset+rows][0],NYPROB,1);
   }

 /* Now call update to update the value of grid points */
 update(start,end,NYPROB,&u[iz][0][0],&u[1-iz][0][0]);
```

```
    iz = 1 - iz;
    }

  /* Finally, send my portion of final results back to parent */
  rc = pvm_initsend(PvmDataRaw);
  rc = pvm_pkint(&offset, 1, 1);
  rc = pvm_pkint(&rows, 1, 1);
  rc = pvm_pkfloat(&u[iz][offset][0],rows*NYPROB,1);
  rc = pvm_send(parent,DONE);

  /* Exit PVM */
  rc = pvm_exit();

  }  /* End of child code */
}  /* End of program */


/*********************  subroutine update *********************/

void update(int start, int end, int ny, float *u1, float *u2) {
int ix, iy;

for (ix = start; ix <= end; ix++)
  for (iy = 1; iy <= ny-2; iy++)
    *(u2+ix*ny+iy) = *(u1+ix*ny+iy)  +
     parms.cx * (*(u1+(ix+1)*ny+iy) + *(u1+(ix-1)*ny+iy) -
                2.0 * *(u1+ix*ny+iy)) +
     parms.cy * (*(u1+ix*ny+iy+1) + *(u1+ix*ny+iy-1) -
                2.0 * *(u1+ix*ny+iy));
```

```
}

/************************* subroutine inidat **********************/
void inidat(int nx, int ny, float *u) {
int ix, iy;

for (ix = 0; ix <= nx-1; ix++)
 for (iy = 0; iy <= ny-1; iy++)
  *(u+ix*ny+iy) = (float)(ix * (nx - ix - 1) * iy * (ny - iy - 1));
}

/********************** subroutine prtdat **************************/
void prtdat(int nx, int ny, float *u1, char *fnam) {
int ix, iy;
FILE *fp;

fp = fopen(fnam, "w");
for (iy = ny-1; iy >= 0; iy--) {
 for (ix = 0; ix <= nx-1; ix++) {
  fprintf(fp, "%6.1f", *(u1+ix*ny+iy));
  if (ix != nx-1)
   fprintf(fp, " ");
  else
   fprintf(fp, "\n");
  }
 }
fclose(fp);
}
```

2) The Makefile and the result:

```
#####################################################################
# FILE: Makefile.aimk
# USE: aimk
#####################################################################

CC    =     cc
OBJ   =     heat2D
SRC   =     pvm_heat2D.c draw_heat.c
PVMDIR =     /mnt/spark2/share/lixin/pvm/unixpvm/pvm3
INCLUDE =    -I${PVMDIR}/include
LIBS  =     -L${PVMDIR}/lib/SUN4SOL2 -lpvm3 -lsocket -lnsl
XLIBS =     /usr/lib/libX11.a

${OBJ}: ${SRC}
        ${CC} ${SRC} ${INCLUDE} ${LIBS} ${XLIBS} -o ${OBJ}


################################### Result ##############################
pvm
pvm> add lily.cs.concordia.ca
Console: exit handler called
1 successful
                HOST    DTID
   lily.cs.concordia.ca    80000
pvm> add orchid.cs.concordia.ca
0 successful
                HOST    DTID
 orchid.cs.concordia.ca Duplicate host
pvm> add spark.cs.concordia.ca
```

```
Console: exit handler called
1 successful
                HOST    DTID
  spark.cs.concordia.ca    c0000
pvm> halt
Terminated


orchid.zhang_j > heat2D
Spawning 3 children processes
...child task id= 262203
...child task id= 262204
...child task id= 262205
Grid size: X= 20  Y= 20  Time steps= 50
Initializing grid and writing initial.dat file...
Sent to= 262203 offset= 0 rows= 7 neighbor1= 0 neighbor2= 262204
Sent to= 262204 offset= 7 rows= 7 neighbor1= 262203 neighbor2= 262205
Sent to= 262205 offset= 14 rows= 6 neighbor1= 262204 neighbor2= 0

0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
  0.0  324.0  612.0  864.0 1080.0 1260.0 1404.0 1512.0 1584.0 1620.0 1620.0 158
4.0 1512.0 1404.0 1260.0 1080.0  864.0  612.0  324.0    0.0
  0.0  612.0 1156.0 1632.0 2040.0 2380.0 2652.0 2856.0 2992.0 3060.0 3060.0 299
2.0 2856.0 2652.0 2380.0 2040.0 1632.0 1156.0  612.0    0.0
  0.0  864.0 1632.0 2304.0 2880.0 3360.0 3744.0 4032.0 4224.0 4320.0 4320.0 422
4.0 4032.0 3744.0 3360.0 2880.0 2304.0 1632.0  864.0    0.0
  0.0 1080.0 2040.0 2880.0 3600.0 4200.0 4680.0 5040.0 5280.0 5400.0 5400.0 528
0.0 5040.0 4680.0 4200.0 3600.0 2880.0 2040.0 1080.0    0.0
  0.0 1260.0 2380.0 3360.0 4200.0 4900.0 5460.0 5880.0 6160.0 6300.0 6300.0 616
```

0.0 5880.0 5460.0 4900.0 4200.0 3360.0 2380.0 1260.0 0.0

0.0 1404.0 2652.0 3744.0 4680.0 5460.0 6084.0 6552.0 6864.0 7020.0 7020.0 686
4.0 6552.0 6084.0 5460.0 4680.0 3744.0 2652.0 1404.0 0.0

0.0 1512.0 2856.0 4032.0 5040.0 5880.0 6552.0 7056.0 7392.0 7560.0 7560.0 739
2.0 7056.0 6552.0 5880.0 5040.0 4032.0 2856.0 1512.0 0.0

0.0 1584.0 2992.0 4224.0 5280.0 6160.0 6864.0 7392.0 7744.0 7920.0 7920.0 774
4.0 7392.0 6864.0 6160.0 5280.0 4224.0 2992.0 1584.0 0.0

0.0 1620.0 3060.0 4320.0 5400.0 6300.0 7020.0 7560.0 7920.0 8100.0 8100.0 792
0.0 7560.0 7020.0 6300.0 5400.0 4320.0 3060.0 1620.0 0.0

0.0 1620.0 3060.0 4320.0 5400.0 6300.0 7020.0 7560.0 7920.0 8100.0 8100.0 792
0.0 7560.0 7020.0 6300.0 5400.0 4320.0 3060.0 1620.0 0.0

0.0 1584.0 2992.0 4224.0 5280.0 6160.0 6864.0 7392.0 7744.0 7920.0 7920.0 774
4.0 7392.0 6864.0 6160.0 5280.0 4224.0 2992.0 1584.0 0.0

0.0 1512.0 2856.0 4032.0 5040.0 5880.0 6552.0 7056.0 7392.0 7560.0 7560.0 739
2.0 7056.0 6552.0 5880.0 5040.0 4032.0 2856.0 1512.0 0.0

0.0 1404.0 2652.0 3744.0 4680.0 5460.0 6084.0 6552.0 6864.0 7020.0 7020.0 686
4.0 6552.0 6084.0 5460.0 4680.0 3744.0 2652.0 1404.0 0.0

0.0 1260.0 2380.0 3360.0 4200.0 4900.0 5460.0 5880.0 6160.0 6300.0 6300.0 616
0.0 5880.0 5460.0 4900.0 4200.0 3360.0 2380.0 1260.0 0.0

0.0 1080.0 2040.0 2880.0 3600.0 4200.0 4680.0 5040.0 5280.0 5400.0 5400.0 528
0.0 5040.0 4680.0 4200.0 3600.0 2880.0 2040.0 1080.0 0.0

0.0 864.0 1632.0 2304.0 2880.0 3360.0 3744.0 4032.0 4224.0 4320.0 4320.0 422
4.0 4032.0 3744.0 3360.0 2880.0 2304.0 1632.0 864.0 0.0

0.0 612.0 1156.0 1632.0 2040.0 2380.0 2652.0 2856.0 2992.0 3060.0 3060.0 299
2.0 2856.0 2652.0 2380.0 2040.0 1632.0 1156.0 612.0 0.0

0.0 324.0 612.0 864.0 1080.0 1260.0 1404.0 1512.0 1584.0 1620.0 1620.0 158
4.0 1512.0 1404.0 1260.0 1080.0 864.0 612.0 324.0 0.0

0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

2. Matrix Multiply DESCRIPTION:

This project is a matrix multiply program. In this project, the data is distributed among the worker tasks who perform the actual multiplication and send back their respective results to the master task.

The code:

1) pvm_mm_master.c:

```
#include <stdio.h>
#include "pvm3.h"              /* PVM version 3.0 include file */

#define NPROC 4                      /* number of PVM worker tasks to spawn */
#define NRA 62                       /* number of rows in matrix A */
#define NCA 15                       /* number of columns in matrix A */
#define NCB 7                  /* number of columns in matrix B */

main() {
int mtid,                      /* PVM task id of master task */
   wtids[NPROC],               /* array of PVM task ids for worker tasks */
   mtype,                  /* PVM message type */
   rows,                   /* rows of matrix A sent to each worker */
   averow, extra, offset,      /* used to determine rows sent to each worker */
   rcode, i, j;            /* misc */
double a[NRA][NCA],                  /* matrix A to be multiplied */
    b[NCA][NCB],               /* matrix B to be multiplied */
    c[NRA][NCB];               /* result matrix C */
char thishost[35];             /* name of selected master */

/* enroll this task in PVM */
mtid = pvm_mytid();
```

```
/* The master task now spawns worker tasks by calling pvm_spawn. The unique
   worker task ids are stored in the wtids array. The first worker task is
   spawned on a specific machine. The return code tells the number of tasks
   successfully spawned, and in this example, is not checked for errors.    */
for (i=0; i<NPROC; i++) {
 if (i==0) {
   printf ("Enter selected hostname - must match PVM config: ");
   scanf("%s", thishost);
   rcode = pvm_spawn("mm.worker", NULL, PvmTaskHost, thishost, 1, &wtids[0]);
   }
 else
   rcode = pvm_spawn("mm.worker", NULL, PvmTaskDefault, "", 1, &wtids[i]);
 }

/* initialize A and B */
for (i=0; i<NRA; i++)
  for (j=0; j<NCA; j++)
    a[i][j]= i+j;

for (i=0; i<NCA; i++)
  for (j=0; j<NCB; j++)
    b[i][j]= i*j;

averow = NRA/NPROC;
extra = NRA%NPROC;
offset = 0;
mtype = 1;

/* send data to the worker tasks */
```

```
 for (i=0; i<NPROC; i++)      {                          /* for each worker task */
  rows = (i < extra) ? averow+1 : averow;     /* Find #rows to send from A */

  /* next call initializes send buffer and specifies to do XDR data format */
  /* conversion only in heterogenous environment */
  rcode = pvm_initsend(PvmDataDefault);

  /* next four calls pack values into the send buffer */
  rcode = pvm_pkint(&offset, 1, 1);            /* starting pos. in matrix */
  rcode = pvm_pkint(&rows, 1, 1);              /* #rows of A to send */
  rcode = pvm_pkdouble(&a[offset][0], rows*NCA, 1); /* some rows from A */
  rcode = pvm_pkdouble(b, NCA*NCB, 1);         /* all of B */

  /* send contents of send buffer to worker task  */
  rcode = pvm_send(wtids[i], mtype);

  offset = offset + rows;
 }

/* wait for results from all worker tasks */
mtype = 2;                                     /* set message type */
for (i=0; i<NPROC; i++)      {                       /* do once for each worker */
  rcode = pvm_recv(-1, mtype);                       /* receive message from
worker*/
  rcode = pvm_upkint(&offset, 1, 1);           /* starting pos. in matrix */
  rcode = pvm_upkint(&rows, 1, 1);             /* #rows sent */
  rcode = pvm_upkdouble(&c[offset][0], rows*NCB, 1); /* rows for matrix C */
  }

/* print results */
```

```
for (i=0; i<NRA; i++) {
  printf("\n");
  for (j=0; j<NCB; j++)
    printf("%6.2f  ", c[i][j]);
  }
printf ("\n");

/* task now exits from PVM */
rcode = pvm_exit();
}
```

2) pvm_mm_worker:

```
#include <stdio.h>
#include <malloc.h>
#include "pvm3.h"      /* PVM version 3.0 include file */

#define NRA 62         /* number of rows in matrix A */
#define NCA 15         /* number of columns in matrix A */
#define NCB 7          /* number of columns in matrix B */

main() {
int wtid,              /* PVM task id of this worker program */
   mtid,               /* PVM task id of parent master program */
   mtype,              /* PVM message type */
   rows,               /* number of rows in matrix a sent to worker */
   offset,             /* starting position in matrix */
   rcode, i, j, k;     /* misc */
double a[NRA][NCA],    /* matrix A to be multiplied */
     b[NCA][NCB],      /* matrix B to be multiplied */
     c[NRA][NCB];      /* result matrix C */
```

```
/* enroll worker task */
wtid = pvm_mytid();

/* Receive message from master */
mtype = 1;                          /* set message type */
mtid = pvm_parent();                /* get task id for master process */
rcode = pvm_recv(mtid, mtype);      /* wait to receive message from master*/
rcode = pvm_upkint(&offset, 1, 1);  /* start pos. in A and C matrices */
rcode = pvm_upkint(&rows, 1, 1);    /* #rows in matrix A sent */
rcode = pvm_upkdouble(a, rows*NCA, 1); /* our share of matrix A */
rcode = pvm_upkdouble(b, NCA*NCB, 1);  /* contents of matrix B */

printf("worker task id = %d received %d rows from A\n", wtid, rows);

/* do matrix multiply */
for (k=0; k<NCB; k++)
 for (i=0; i<rows; i++) {
  c[i][k] = 0.0;
  for (j=0; j<NCA; j++)
   c[i][k] = c[i][k] + a[i][j] * b[j][k];
 }

/* Set up send message to master. */
mtype = 2;                          /* set message type */
rcode = pvm_initsend(PvmDataDefault);  /* initialize send buffer */
rcode = pvm_pkint(&offset, 1, 1);   /* pos. in result matrix */
rcode = pvm_pkint(&rows, 1, 1);     /* number of rows being sent */
rcode = pvm_pkdouble(c, rows*NCB, 1);  /* our part of result matrix C */
```

```
/* send to master */
rcode = pvm_send(mtid, mtype);

/* exit PVM */
rcode = pvm_exit();
}
```

3) The Makefile and result:

```
##########################################################################
# PVM Matrix Multiply Makefile
# USE: aimk
##########################################################################

CC      =       cc
PVMDIR  =    /mnt/spark2/share/lixin/pvm/unixpvm/pvm3
INCLUDE =    -I${PVMDIR}/include
LIBS    =    -L${PVMDIR}/lib/SUN4SOL2 -lpvm3 -lsocket -lnsl


mm:    mm.master mm.worker

mm.master:    pvm_mm_master.c
        ${CC} pvm_mm_master.c ${INCLUDE} ${LIBS} -o mm.master

mm.worker:    pvm_mm_worker.c
        ${CC} pvm_mm_worker.c ${INCLUDE} ${LIBS} -o mm.worker



orchid.zhang_j > mm.master
```

Enter selected hostname - must match PVM config: orchid

0.00	1015.00	2030.00	3045.00	4060.00	5075.00	6090.00
0.00	1120.00	2240.00	3360.00	4480.00	5600.00	6720.00
0.00	1225.00	2450.00	3675.00	4900.00	6125.00	7350.00
0.00	1330.00	2660.00	3990.00	5320.00	6650.00	7980.00
0.00	1435.00	2870.00	4305.00	5740.00	7175.00	8610.00
0.00	1540.00	3080.00	4620.00	6160.00	7700.00	9240.00
0.00	1645.00	3290.00	4935.00	6580.00	8225.00	9870.00
0.00	1750.00	3500.00	5250.00	7000.00	8750.00	10500.00
0.00	1855.00	3710.00	5565.00	7420.00	9275.00	11130.00
0.00	1960.00	3920.00	5880.00	7840.00	9800.00	11760.00
0.00	2065.00	4130.00	6195.00	8260.00	10325.00	12390.00
0.00	2170.00	4340.00	6510.00	8680.00	10850.00	13020.00
0.00	2275.00	4550.00	6825.00	9100.00	11375.00	13650.00
0.00	2380.00	4760.00	7140.00	9520.00	11900.00	14280.00
0.00	2485.00	4970.00	7455.00	9940.00	12425.00	14910.00
0.00	2590.00	5180.00	7770.00	10360.00	12950.00	15540.00
0.00	2695.00	5390.00	8085.00	10780.00	13475.00	16170.00
0.00	2800.00	5600.00	8400.00	11200.00	14000.00	16800.00
0.00	2905.00	5810.00	8715.00	11620.00	14525.00	17430.00
0.00	3010.00	6020.00	9030.00	12040.00	15050.00	18060.00
0.00	3115.00	6230.00	9345.00	12460.00	15575.00	18690.00
0.00	3220.00	6440.00	9660.00	12880.00	16100.00	19320.00
0.00	3325.00	6650.00	9975.00	13300.00	16625.00	19950.00
0.00	3430.00	6860.00	10290.00	13720.00	17150.00	20580.00
0.00	3535.00	7070.00	10605.00	14140.00	17675.00	21210.00
0.00	3640.00	7280.00	10920.00	14560.00	18200.00	21840.00
0.00	3745.00	7490.00	11235.00	14980.00	18725.00	22470.00
0.00	3850.00	7700.00	11550.00	15400.00	19250.00	23100.00

0.00	3955.00	7910.00	11865.00	15820.00	19775.00	23730.00
0.00	4060.00	8120.00	12180.00	16240.00	20300.00	24360.00
0.00	4165.00	8330.00	12495.00	16660.00	20825.00	24990.00
0.00	4270.00	8540.00	12810.00	17080.00	21350.00	25620.00
0.00	4375.00	8750.00	13125.00	17500.00	21875.00	26250.00
0.00	4480.00	8960.00	13440.00	17920.00	22400.00	26880.00
0.00	4585.00	9170.00	13755.00	18340.00	22925.00	27510.00
0.00	4690.00	9380.00	14070.00	18760.00	23450.00	28140.00
0.00	4795.00	9590.00	14385.00	19180.00	23975.00	28770.00
0.00	4900.00	9800.00	14700.00	19600.00	24500.00	29400.00
0.00	5005.00	10010.00	15015.00	20020.00	25025.00	30030.00
0.00	5110.00	10220.00	15330.00	20440.00	25550.00	30660.00
0.00	5215.00	10430.00	15645.00	20860.00	26075.00	31290.00
0.00	5320.00	10640.00	15960.00	21280.00	26600.00	31920.00
0.00	5425.00	10850.00	16275.00	21700.00	27125.00	32550.00
0.00	5530.00	11060.00	16590.00	22120.00	27650.00	33180.00
0.00	5635.00	11270.00	16905.00	22540.00	28175.00	33810.00
0.00	5740.00	11480.00	17220.00	22960.00	28700.00	34440.00
0.00	5845.00	11690.00	17535.00	23380.00	29225.00	35070.00
0.00	5950.00	11900.00	17850.00	23800.00	29750.00	35700.00
0.00	6055.00	12110.00	18165.00	24220.00	30275.00	36330.00
0.00	6160.00	12320.00	18480.00	24640.00	30800.00	36960.00
0.00	6265.00	12530.00	18795.00	25060.00	31325.00	37590.00
0.00	6370.00	12740.00	19110.00	25480.00	31850.00	38220.00
0.00	6475.00	12950.00	19425.00	25900.00	32375.00	38850.00
0.00	6580.00	13160.00	19740.00	26320.00	32900.00	39480.00
0.00	6685.00	13370.00	20055.00	26740.00	33425.00	40110.00
0.00	6790.00	13580.00	20370.00	27160.00	33950.00	40740.00
0.00	6895.00	13790.00	20685.00	27580.00	34475.00	41370.00
0.00	7000.00	14000.00	21000.00	28000.00	35000.00	42000.00

0.00	7105.00	14210.00	21315.00	28420.00	35525.00	42630.00
0.00	7210.00	14420.00	21630.00	28840.00	36050.00	43260.00
0.00	7315.00	14630.00	21945.00	29260.00	36575.00	43890.00
0.00	7420.00	14840.00	22260.00	29680.00	37100.00	44520.00

3. Simple Array DESCRIPTION:

This project is an array assignment used to demonstrate the distribution of data among multiple tasks and the communications required to accomplish it. The master task distributes an equal portion of the array to each worker task. Each worker task receives its portion of the array and performs a simple value assignment to each of its elements. Each worker then sends its portion of the array back to the master. As the master receives back each portion of the array selected elements are displayed.

1) pvm_arrary_master.c:

```
#include <stdio.h>
#include "pvm3.h"                    /* PVM version 3.0 include file */

#define     NTASKS          6
#define     ARRAYSIZE 600000
#define     FROMMASTER_MSG      1
#define     FROMWORKER_MSG      2
#define WORKERTASK      "array.worker"

main() {
int     tids[NTASKS],
        rc,         /* for catching PVM return codes */
        i,          /* loop variable */
        index,      /* index into the array */
        tid,        /* PVM task id */
        bufid,      /* PVM message buffer id */
        bytes,      /* number bytes recv'd in PVM message buffer */
        msgtype,    /* PVM message type */
```

```
        chunksize;     /* for partitioning the array */
float   data[ARRAYSIZE], /* the intial array */
        result[ARRAYSIZE];  /* for holding results of array operations */

/************************* enroll this task in PVM ************************
* pvm_mytid will enroll this process in your PVM virtual machine. A unique
* task id will be assigned if call is successful. The pvmds keep track of
* processes and communications via task ids. Return codes less than zero
* indicate an error in the enroll process and will terminate this program.
**************************************************************************/
printf("\n************ Starting PVM Example ************\n");
rc = pvm_mytid();
if (rc < 0) {
 printf("MASTER: Unable to enroll this task.\n");
 printf("   Enroll return code= %d.  Quitting.\n", rc);
 exit(0);
 }
else
 printf("MASTER: Enrolled as task id = %d\n", rc);

/************************* spawn worker tasks ****************************
* The master task now spawns the worker tasks by calling pvm_spawn. The unique
* task ids for workers are stored in the tids array. The return code tells
* the number of tasks successfully spawned. In this example, it must equal
* NTASKS, otherwise the program terminates.
**************************************************************************/
printf("MASTER: Spawning worker tasks...\n");
rc = pvm_spawn(WORKERTASK, NULL, PvmTaskDefault, "", NTASKS, tids);
if (rc == NTASKS)
 printf("MASTER: Successfully spawned %d worker tasks.\n", rc);
```

```
else {
 printf("MASTER: Not able to spawn requested number of tasks!\n");
 printf("MASTER: Tasks actually spawned: %d.  Quitting.\n",rc);
 exit(0);
 }

/************************ initializations ********************************
* Define the partition size as chunksize and then initialize the array to 0
************************************************************************/
chunksize = (ARRAYSIZE / NTASKS);
for(i=0; i<ARRAYSIZE; i++)
 data[i] =  0.0;

/******************* send array chunks to each worker task *****************
* Data passing from master task to each worker task begins by initializing the
* send buffer with a call to pvm_initsend.  Its argument tells PVM that XDR
* data conversion should be performed only if the virtual machine is hetero
* genous.  Data is packed sequentially into the buffer with the calls to
* pvm_pkint and pvm_pkfloat.  The arguments to these calls specify the data
* address, number of values and stride.  Each worker is sent the following:
*   index = the starting index for this worker's partition of the array;
*         one value is sent with a stride on one.
*   chunksize = the partition size of the array, one value is sent with a
*         stride of one.
*   data[] = the actual data for this worker's array partition; chunksize
*         number of values are sent with a stride of one.
* Finally, the data is sent to each worker task by calling pvm_send.  Its
* arguments specify each task id that is to receive the data and which
* message type should be set.  Index is incremented by chunksize for the
* next workers partition of the array.  Note that PVM routine calls provide
```

```
* return codes even if they are not always used in this program.
**************************************************************************/
printf("MASTER: Sending data to worker tasks...\n");
index = 0;
msgtype = FROMMASTER_MSG;
for (i=0; i<NTASKS; i++) {
 rc = pvm_initsend(PvmDataDefault);
 rc = pvm_pkint(&index, 1, 1);
 rc = pvm_pkint(&chunksize, 1, 1);
 rc = pvm_pkfloat(&data[index], chunksize, 1);
 rc = pvm_send(tids[i], msgtype);
 index = index + chunksize; }

/***************** wait for results from all worker tasks ******************
* The master task now waits in a loop to receive each worker's partition of
* the array. pvm_recv blocks until it receives a message of type "msgtype"
* from task id = -1, which means, "any task id". pvm_recv returns the PVM
* message buffer id of the awaited message as bufid. Bufid is then used to
* find out additional information about the message by calling pvm_bufinfo.
* The data in the message buffer is unpacked sequentially with calls to
* pvm_upkint and pvm_upkfloat. The arguments to pvm_upk* calls specify the
* address of the data, number of values and stride. The master knows which
* part of the result array it is receiving by the value of index. As above
* the size of the array partition is determined by chunksize. Note that it
* is the programmer's responsibility to insure that message types and data
* sequences match between data sends/receives. Finally, the master task
* prints a sample of the returned result partition and also tells which
* task id it came from by the value of tid, as returned from the call to
* pvm_bufinfo.
**************************************************************************/
```

```
printf("MASTER: Waiting for results from worker tasks...\n");
msgtype = FROMWORKER_MSG;
for(i=0; i<NTASKS; i++){
  bufid = pvm_recv(-1, msgtype);
  rc = pvm_bufinfo(bufid, &bytes, &msgtype, &tid);
  rc = pvm_upkint(&index, 1, 1);
  rc = pvm_upkfloat(&result[index], chunksize, 1);
  printf("-------------------------------------------------\n");
  printf("MASTER: Sample results from worker task = %d\n",tid);
  printf("   result[%d]=%f\n", index, result[index]);
  printf("   result[%d]=%f\n", index+100, result[index+100]);
  printf("   result[%d]=%f\n\n", index+1000, result[index+1000]);
  }

/************************** exit from PVM ******************************
* Leave PVM before program finishes.  This is "good programming practice"
* since it allows all of the pvmds in your virtual machine to keep track of
* which tasks are active and which are not.  A return code less than zero
* from pvm_exit indicates an error which is ignored by this program.
***********************************************************************/
printf("MASTER: All Done! \n");
rc = pvm_exit();

}
```

2) pvm_array_worker.c

```
#include     <stdio.h>
#include     "pvm3.h"      /* PVM version 3.0 include file */
```

```
#define ARRAYSIZE       600000
#define FROMMASTER_MSG  1
#define FROMWORKER_MSG  2

main() {
int    masterid,            /* PVM task id for master process */
       rc,                  /* for catching PVM return codes */
       i,                   /* loop variable */
       index,               /* index into the array */
       msgtype,             /* PVM message type */
       chunksize;           /* for partitioning the array */
float  result[ARRAYSIZE];   /* for holding results of array operations */


/********************* enroll this task in PVM ****************************
* pvm_mytid will enroll this process in your PVM virtual machine.  A unique
* task id will be assigned if call is successful. The pvmds keep track of
* processes and communications via task ids. Return codes less than zero
* indicate an error in the enroll process and will terminate this program.
* Note that worker programs, though spawned from the master, must still enroll.
*************************************************************************/
rc = pvm_mytid();
if (rc < 0) {
 printf("WORKER: Unable to enroll this task.\n");
 printf("  Enroll return code= %d.  Quitting.\n", rc);
 exit(0);
 }
else {
 printf("WORKER: Enrolled as task id = %d\n", rc);
 }
```

```
/******************** receive data from master task ************************
* The worker task waits/blocks here until it receives a message from its
* parent - the master task - of the correct message type. It knows the task
* id of its parent from the call to pvm_parent. When the correct message
* is recieved, the data is unpacked by calls to pvm_upkint and pvm_upkfloat
* in the identical sequential order that it was originally sent from the
* master. Note that it is the programmer's responsibility to insure that
* message types and data sequences match between data sends/receives.
*****************************************************************************/
msgtype = FROMMASTER_MSG;
masterid = pvm_parent();
rc = pvm_recv(masterid, msgtype);
rc = pvm_upkint(&index, 1, 1);
rc = pvm_upkint(&chunksize, 1, 1);
rc = pvm_upkfloat(&result[index], chunksize, 1);

/*************** modify the array before sending it back to master ************
* Real complex operation going on here - the worker just adds one to the
* index value of the array location.
*****************************************************************************/
for(i=index; i < index + chunksize; i++)
 result[i] = i + 1;


/******************* send results back to master task ************************
* Data passing from this worker task to master task begins by initializing the
* send buffer with a call to pvm_initsend. Its argument tells PVM that XDR
* data conversion should be performed only if the virtual machine is hetero
* genous. Data is packed sequentially into the buffer with the calls to
```

```
* pvm_pkint and pvm_pkfloat. The arguments to these calls specify the data
* address, number of values and stride. Each worker sends the following:
*   index = the starting index for this worker's partition of the array;
*         one value is sent with a stride on one.
*   result[] = the actual data for this worker's array partition; chunksize
*         number of values are sent with a stride of one.
* Finally, the data is sent by the worker task by calling pvm_send. Its
* arguments specify the task id of the task which should receive the data -
* in this case the master task id which it obtained previously - and which
* message type should be set. Note that PVM routine calls provide return
* codes even if they are not always used in this program.
**************************************************************************/
msgtype = FROMWORKER_MSG;
rc = pvm_initsend(PvmDataDefault);
rc = pvm_pkint(&index, 1, 1);
rc = pvm_pkfloat(&result[index], chunksize, 1);
rc = pvm_send(masterid, msgtype);

/*************************** exit from PVM ********************************
* Leave PVM before program finishes. This is "good programming practice"
* since it allows all of the pvmds in your virtual machine to keep track of
* which tasks are active and which are not. A return code less than zero
* from pvm_exit indicates an error which is ignored by this program.
**************************************************************************/
rc = pvm_exit();  }
```

3) Makefile and results:

```
##########################################################################
# PVM Makefile - Array Example
# USE: aimk
##########################################################################

CC      =       cc
MASTER      =       array.master
MASTERSRC =       pvm_array_master.c
WORKER      =       array.worker
WORKERSRC =       pvm_array_worker.c
PVMDIR   =    /mnt/spark2/share/lixin/pvm/unixpvm/pvm3
INCLUDE  =    -I${PVMDIR}/include
LIBS     =    -L${PVMDIR}/lib/SUN4SOL2 -lpvm3 -lsocket -lnsl

all:     ${MASTER} ${WORKER}

${MASTER}: ${MASTERSRC}
        ${CC} ${MASTERSRC} ${INCLUDE} ${LIBS} -o ${MASTER}

${WORKER}: ${WORKERSRC}
        ${CC} ${WORKERSRC} ${INCLUDE} ${LIBS} -o ${WORKER}

##########################################################################
orchid.zhang_j > array.master

********** Starting PVM Example **********
MASTER: Enrolled as task id = 262195
MASTER: Spawning worker tasks...
```

```
MASTER: Successfully spawned 2 worker tasks.
MASTER: Sending data to worker tasks...
MASTER: Waiting for results from worker tasks...
-----------------------------------------------
MASTER: Sample results from worker task = 262196
  result[0]=1.000000
  result[100]=101.000000
  result[1000]=1001.000000

-----------------------------------------------
MASTER: Sample results from worker task = 262197
  result[100000]=100001.000000
  result[100100]=100101.000000
  result[101000]=101001.000000

MASTER: All Done!
```

4. Timing DESCRIPTION:

This project is a PVM communication timing test. The master program will send "reps" number of messages to the worker task, waiting for a reply between each rep. Before and after timings are made for each rep and an average calculated when completed.

1) timing.c

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include "pvm3.h"              /* PVM version 3.0 include file */

#define WORKERNAME "timing.worker"   /* name of PVM worker executable */

main() {
        int mytid;                   /* PVM task id for master */
        int wtid;                    /* PVM task id for worker */
        int rcode;                   /* PVM return code */
        int reps = 20;               /* number of samples per test */
        struct timeval tv1, tv2;     /* for timing */
        int dt1, dt2;                /* time for one iter */
        int at1, at2;                /* accum. time */
        int n;
        int onevalue = 0;            /* minimal message to send */

        /* attempt to enroll this task in PVM - quit if enroll fails */
        mytid = pvm_mytid();
        if (mytid < 0) {
```

```
            fputs("can't enroll\n", stderr);
            exit(0);
        }

        /* attempt to spawn worker task - quit if spawn fails */
        rcode = pvm_spawn(WORKERNAME, NULL, PvmTaskDefault, "", 1,
&wtid);
        if (rcode < 0) {
            fprintf(stderr, "Can't spawn %s. Quitting.\n", WORKERNAME);
            goto bail;
        }

        /* round-trip timing test */
        printf("Doing round trip test, minimal message size, %d reps.\n",reps);
        at1 = 0;

        /* next call initializes the PVM send buffer */
        rcode = pvm_initsend(PvmDataDefault);

        /* pack a single dummy value into the send message buffer */
        rcode = pvm_pkint(&onevalue, 1, 1);

        for (n = 1; n <= reps; n++) {

            gettimeofday(&tv1, (struct timeval*)0);        /* before time */

            /* send message to worker - message type set to 1.  If  */
            /* return code is less than zero quit */
            rcode = pvm_send(wtid, 1);
            if (rcode < 0) {
```

```
                fprintf(stderr, "Can't send to %s\n", WORKERNAME);
                goto kbail;
        }

        /* Now wait to receive the echo reply from the worker  */
        /* This message type is set to 2.  Quit if return code */
        /* is less than zero */
        rcode = pvm_recv(wtid, 2);
        if (rcode < 0) {
                fprintf(stderr, "Recv error from %s\n", WORKERNAME);
                goto kbail;
        }

        gettimeofday(&tv2, (struct timeval*)0);  /* after time */

        /* calculate round trip time and print */
        dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
            - tv1.tv_usec;
        printf("round trip# %2d   uSec = %8d\n", n, dt1);
        at1 += dt1;
    }
    printf("\n*** Round Trip Avg uSec = %d\n", at1 / reps);

kbail:
    /* need to explicitly kill worker process because it operates in */
    /* an infinite loop */
    rcode = pvm_kill(wtid);
bail:
    /* now exit from PVM cleanly before quitting */
    rcode = pvm_exit();
```

```
        exit(0);
}
```

2) timing_worker.c:

```
#include <stdio.h>
#include "pvm3.h"           /* PVM version 3.0 include file */

main() {
        int mytid;              /* PVM process task id */
        int parent;             /* PVM process task id for parent */
        int rcode;              /* PVM return code */
        int n = 0;
        int onevalue = 0;       /* minimal message to send */

        mytid = pvm_mytid();/* enroll worker task in PVM virtual machine */
        parent = pvm_parent();          /* get PVM task id of parent/master process */

        /* next call initializes the PVM send buffer */
        rcode = pvm_initsend(PvmDataDefault);

        /* pack a single dummy value into the send message buffer */
        rcode = pvm_pkint(&onevalue, 1, 1);

        /* infinite loop - this task must be explicitly killed by parent */
        while (1) {
                /* wait for message of type 1 from parent/master task */
                rcode = pvm_recv(parent, 1);

                /* immediately reply back to parent/master after receive */
                rcode = pvm_send(parent, 2);
```

```
            printf("   worker task echo %d\n", ++n);
        fflush(stdout);
      }
}
```

3) results:

```
orchid.zhang_j > timing
i'm t4005f
slave is task t40060
Doing Round Trip test, minimal message size

 N    uSec
 1    641
 2    557
 3    493
 4    490
 5    490
 6    491
 7    494
 8    491
 9    762
10    519
11    493
12    491
13    490
14    488
15    488
16    491
```

17 488

18 489

19 489

20 489

RTT Avg uSec 516

Doing Bandwidth tests

Message size 100

N	Pack uSec	Send uSec
1	38	515
2	36	702
3	46	599
4	39	536
5	45	538
6	35	517
7	36	510
8	36	511
9	36	507
10	36	510
11	36	505
12	36	504
13	37	506
14	37	506
15	36	510
16	37	512
17	35	509
18	36	508
19	36	508

20 37 800

Avg uSec

37 540

Avg Byte/uSec

2.702703 0.185185

Message size 1000

N	Pack uSec	Send uSec
1	50	617
2	46	587
3	43	523
4	44	530
5	42	525
6	43	528
7	42	527
8	44	527
9	42	526
10	43	527
11	42	525
12	43	525
13	43	900
14	45	541
15	43	522
16	44	524
17	43	523
18	45	523
19	44	525
20	44	525

Avg uSec

43 552

Avg Byte/uSec

23.255814 1.811594

Message size 10000

N	Pack uSec	Send uSec
1	318	1339
2	121	1088
3	120	1437
4	123	1100
5	121	1362
6	145	1193
7	120	1073
8	117	1076
9	118	1064
10	118	1075
11	223	1157
12	120	1082
13	120	1066
14	120	1071
15	120	1073
16	118	1072
17	119	1067
18	119	1162
19	120	1071
20	117	1073

Avg uSec

135 1135

Avg Byte/uSec

74.074074 8.810573

Message size 100000

N	Pack uSec	Send uSec
1	4038	10174
2	897	7123
3	967	7487
4	886	7650
5	892	7464
6	1016	7245
7	917	7888
8	1080	7714
9	887	8262
10	890	7555
11	900	7597
12	983	8041
13	906	7252
14	886	7154
15	1054	6910
16	880	7377
17	891	7655
18	889	7531
19	888	7752
20	886	7808
Avg uSec		
	1081	7681
Avg Byte/uSec		
	92.506938	13.019138

Message size 1000000

N	Pack uSec	Send uSec
1	44960	92243

```
 2      8888    151336
 3      9104     69529
 4      9272     74426
 5      9283     74869
 6     11311     73559
 7      8852     66958
 8      8839    144735
 9      8885     73547
10      8956     74026
11      8834     73511
12      8757     73391
13      8872     74216
14      8880     73133
15      8744     74337
16      8831     74710
17      9200     74971
18      8838     75309
19      9126     75072
20      9151     70390
Avg uSec
       10879     81713
Avg Byte/uSec
    91.920213   12.237955

done
orchid.zhang_j >
```

5. Master-Slave DESCRIPTION:

This is a Master-Slave program. Its first PVM action is to obtain the task id of the ``master" using the pvm_parent call. This program then obtains its num_data and calculating it first. Then it will transmit it to the master using the three-call sequence - pvm_initsend to initialize the send buffer; pvm_pkint to place an integer, in a strongly typed and architecture-independent manner, into the send buffer; and pvm_send to transmit it to the destination process specified by *master*, ``tagging" the message with the number 7.

1) master.c:

2)

```
/*******************************************/
/********* Learner's  Master.c *****************/
/*******************************************/

# include "/mnt/spark2/share/lixin/pvm/unixpvm/pvm3/include/pvm3.h"
#include <stdio.h>
#define SIZE 1000
#define Nprocs 5

main()
{
int mytID task_ids[Nprocs];
int a[SIZE], result[Nprocs], sum=0;
int i, msgType, num_data=SIZE/Nprocs;


/*enroll in PVM*/

mytID=pvm_mytid();
```

```
for(i=o; i< SIZE; i++)
        a[i]=i%25;
/*spawn worker tasker*/
pvm_spawn("worker", (char**)0,  PvmTaskDefault, "", Nprocs, task_ids);
/*send data to worker tasks */
for(i=0;i<Nprocs; i++)
{
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&num_data, 1, 1);
        pvm_pkint(&a[num_data*i], num_data, 1);
        pvm_send(task_ids[i], 4);
}

/*wait and gather results */

msgType = 7;
for (i=0; i<Nprocs; i++)

pvm_recv(task_ids[i], msgType);
pvm_upkint(&results[i], 1, 1);

sum+=results[i]; }

printf("The sum is %d\n", sum);

pvm_exit();
}
```

2) slave.c:

```
# include "/mnt/spark2/share/lixin/pvm/unixpvm/pvm3/include/pvm3.h"
#include <stdio.h>

main()
{
int mytID;
int i, sum, *n;
int master, num_data;
/*enroll in PVM*/
mytID=pvm_mytid();

/*receive data from host */
pvm_recv(-1, -1);
pvm_upkint(&num_data, 1, 1);
a=(int*)malloc(num_data*sizeof(int));
pvm_pkint(a, num_data, 1);
sum=0;
for(i=o; i< num_data; i++)
        sum+=a[i];
/* send computed sum back to host*/
master=pvm_parent();
pvm_initsend(PvmDataRaw);
pvm_pkint(&sum, 1, 1);
pvm_send(master,  7);

pvm_exit();
}
```

Appendix D PVM Supported Architectures/Oss

PVM (Parallel Virtual Machine) is a software package that permits a heterogeneous collection of Unix and NT computers hooked together by a network to be used as a single large parallel computer. Thus large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers. With thousands of users, PVM has become the de facto standard for heterogeneous cluster computing world-wide. The source is available free thru netlib and has been compiled on everything from laptops to CRAYs.

Targeted Platforms:

PVM software is very portable. It has been used on all the follow systems. A virtual computer can be composed of a mixture of any of the these computers.

PCs

Pentium II, Pentium Pro, Pentium, Duals and Quads	Win95, NT 3.5.1, NT 4.0 **Linux**, Solaris, SCO, NetBSD, BSDI, FreeBSD
MAC	NetBSD
Next	
Amiga	NetBSD

Workstations and Shared-memory Servers

SUN3, SUN4, SPARC, UltraSPARC	SunOS, Solaris 2.x
IBM RS6000, J30	AIX 3.x, AIX 4.x
HP 9000	HPux
DEC Alpha, Pmax, microvax	OSF, NT-Alpha
SGI	IRIS 5.x, IRIS 6.x

Parallel Computers

Cray YMP, T3D, T3E, Cray2
Convex Exemplar
IBM SP2, 3090
NEC SX-3
Fujitsu
Amdahl
TMC CM5
Intel Paragon
Sequent Symmetry, Balance

XPVM was designed to simplify running and debugging parallel applications