# Getting Started with PVM

Jennifer Schopf*

April 15, 2000

## 1 Introduction

This is a brief guide on how to get started using PVM. These notes are based primarily on a previous version of this document by Eric Webster, and information contained in the following documents:

- PVM Home page
  `http://www.epm.ornl.gov/pvm/pvm_home.html`
- The PVM book
  `http://www.netlib.org/pvm3/book/pvm-book.html`
- PVM 3 User's Guide and Reference Manual
  `http://www.netlib.org/pvm3/ug.ps`
- Intro to Programming on PVM
  `http://www.epm.ornl.gov/pvm/intro.html`

Please think twice before printing out most of the above documents - they're rather long, and the online material is very good.

## 2 System Overview

PVM (Parallel Virtual Machine) is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. The individual computers may be shared- or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations, that may be interconnected by a variety of networks, such as ethernet, FDDI, etc. PVM support software executes on each machine in a user-configurable pool, and presents a unified, general, and powerful computational environment of concurrent applications. User programs written in C or Fortran are provided access to PVM through the use of calls to PVM library routines for functions such as process initiation, message transmission and reception, and synchronization via barriers or rendezvous. Users may optionally control the execution location of specific

*updated by Derrick Kondo

application components. The PVM system transparently handles message routing, data conversion for incompatible architectures, and other tasks that are necessary for operation in a heterogeneous, network environment.

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. The PVM system has been used for a applications such as molecular dynamics simulations, superconductivity studies, distributed fractal computations, matrix algorithms, and in the classroom as the basis for teaching concurrent computing.

PVM version 3.4.3 is the version running on the CSE APE lab machines for which this document is written.

# 3 Logging in for the First Time

Some modifications to your computing environment are required before you can use PVM on a network of workstations. These are somewhat complicated, and need to be followed closely to have your files wind up in the right places. A `different type style` will be used to denote the machine commands or file names involved. If these lines are preceded by a `%` then they are to be typed at the Unix command line. Commands preceded by `pvm>` are to be used as a part of the PVM console.

For the APE lab setup, the PVM files are located in `/home/solaris/ieng9/cs160s/public/pvm3`. Define an environment variable for the directory so you won't have to retype the whole directory name for each command, for example:

```
% setenv PVMDIR /home/solaris/ieng9/cs160s/public/pvm3
```

Now, logged into any machine on the network you will be using, create a new directory called `pvm3` in your home directory. Enter that directory, and make another directory entitled `bin`. Remaining in your `pvm3` directory, type in the following command to create a soft link named `lib` to the master `pvm3/lib` directory:

```
% ln -s $PVMDIR/lib lib
```

Also create a soft link name `conf` to the master `pvm3/conf` directory:

```
% ln -s $PVMDIR/conf conf
```

Then, `cd` into your `pvm3/bin` directory and create a directory for the architecture you will be using on your network. Create a directory named `SUN4SOL2`, since this is the architecture of the uAPE machines on which PVM will be running. Now, `cd` into this directory, `pvm3/bin/SUN4SOL2`, and type the following:

```
% cp $PVMDIR/bin/SUN4SOL2/pvmgs pvmgs
```

This copies a small executable file to your `pvm3/bin/SUN4SOL2/` directory, which is needed if you plan on using the dynamic process group feature of PVM. Now return to your `pvm3` directory, and type:

```
%  cp -r $PVMDIR/include .
```

Note the trailing period on the command. This copies the C include file (and also the fortran include file) that is needed in all PVM source code. All the PVM examples are set up to expect these include files to be in your own directory space.

Now, return to your home directory where you will need to modify your `.cshrc` and `.login` file. These instructions assume that you are either using csh or tcsh. The modification consists of appending the following to your .cshrc file:

```
# the pvm directory
setenv PVM_ROOT /home/solaris/ieng9/cs160s/public/pvm3

# a stub that sets the PVM_ARCH environment variable among others
source /home/solaris/ieng9/cs160s/public/pvm3/lib/cshrc.stub
```

and the following to your .login file:

```
#to allow "man pvm", "man aimk", etc.
if (! $?MANPATH) setenv MANPATH /usr/man:/usr/local/man
setenv MANPATH "${MANPATH}:/home/solaris/ieng9/cs160s/public/pvm3/man"

#to set the path to many of pvm's scipts
setenv PATH /home/solaris/ieng9/cs160s/public/pvm3/lib:$PATH
```

To get these changes in your shell setup to take place, `source` your `.cshrc` files and `.login` files in this order, and `rehash` as well for good measure using the following commands:

```
%  source .cshrc
%  source .login
%  rehash
```

Because PVM needs permission to start processes on remote machines, it is convenient to have a `.rhosts` file in your home directory that contains all of the machines on which you plan to run PVM. For example, a `.rhosts` file for the APE machines could include the following text:

```
ieng9.ucsd.edu Your_Login_Name
uape-1.ucsd.edu Your_Login_Name
uape-2.ucsd.edu Your_Login_Name
uape-3.ucsd.edu Your_Login_Name
uape-4.ucsd.edu Your_Login_Name
uape-5.ucsd.edu Your_Login_Name
uape-6.ucsd.edu Your_Login_Name
uape-7.ucsd.edu Your_Login_Name
uape-8.ucsd.edu Your_Login_Name
uape-9.ucsd.edu Your_Login_Name
uape-10.ucsd.edu Your_Login_Name
```

where `Your_User_Name` is your login name on the APE machines. If you already have a `.rhosts` file that contains other machine names, you can simply add this list to it.

Note that it is advised to protect the contents of your `.rhosts` file by removing all access privilege for *group* and *other*. You can do this with the following command:

```
chmod go-rwx .rhosts
```

After setting up your `.rhosts` file you should now be ready to run PVM!

# 4 Running a Program Using PVM

## 4.1 The PVM Console

The first thing to do is to make sure your PVM environment is ready to go. So what we need to do is to start PVM and to configure a virtual machine. The PVM console, called `pvm`, is a stand-alone PVM task that allows the user to interactively start, query or modify the virtual machine. The console may be started and stopped multiple times on any of the hosts in the virtual machine without affecting PVM or any applications that may be running. On any host on which PVM has been installed you can type:

```
% pvm
```

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

```
pvm>  add hostname
```

where `hostname` is any host in your `.rhost` file. The host you are on is automatically a part of the virtual machine. You can delete hosts (except for the one you are on) from your virtual machine by typing:

```
pvm>  delete hostname
```

If you get a message "Can't Start pvmd," then check the common startups problem section in the PVM book (`http://www.netlib.org/pvm3/book/pvm-book.html`) and try again.

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

```
pvm>  conf
```

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

```
pvm>  ps -a
```

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

```
%  pvm
```

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

```
pvm>  halt
```

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

A full description of the PVM console and it's many command options is given at the end of chapter 3 in the PVM book located on the web at
`http://www.netlib.org/pvm3/book/pvm-book.html`.

## 4.2 Compiling and Running a Program

This section will give you a basic overview on how to compile programs for PVM, including running an example program. Compiling for PVM is very specific on your environment. The following is based on the assumption that you have followed all the setup commands above, and have the files and directories in their proper places.

To use PVM, all files must include the `pvm3.h` file, so all C programs must have the following line in them:

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

To compile and execute a program, you need to have the source code and makefiles in your own `pvm3` directory. If these files are anywhere else, they will not be able to run correctly. Now we'll get the set of example programs supplied by the PVM people. Change into your `pvm3` directory and copy the example directory with the following command:

```
cp -r $PVMDIR/examples .
```

Please note the trailing period at the end of the command.

In your newly made `examples` directory, you'll find the files `Makefile.aimk` and `README` which describe how to build the example programs. PVM3 supplies an architecture-independent make, `aimk`, that automatically determines `PVM_ARCH` (SUN4SOL2 in our case) and links any operating system specific libraries to your application. `aimk` was automatically added to your `$PATH` when you extended your `.login` and `.cshrc` file. Using `aimk` allows you to leave the source code and makefile unchanged as you compile across different architectures.

In particular, the environment variables `PVM_ARCH` and `ARCHLIB`, among other things, are set for you if you use aimk instead of make.

The master/slave programming model is the most popular model used in distributed computing, so we'll look at that as an example. To compile the master/slave C example type:

```
%  aimk master1 slave1
```

The makefile moves the executables to your `pvm3/bin/SUN4SOL2` directory, which is the default location PVM will look for them on all hosts. This is one reason why you need to have followed the startup directions closely.

Now, from one window, start PVM and configure some hosts (see section 4.1 for details). All the example programs are designed to run on any number of hosts, including one. In another window, `cd` to your `pvm3/bin/SUN4SOL2` directory, and type:

```
%  master1
```

The program will spawn 3*(num hosts in the virtual machine) worker tasks. Note that the number of tasks does not have to match the number of hosts in the examples.

This example illustrates the ability to run a PVM program from a Unix prompt on any host in the virtual machine. This is just like the way you would run a serial `a.out` program on a workstation.

# 5 Parallel Programming Primitives

The following section will describe some of the basic programming primitives needed when you use PVM. All PVM routines are written in C, but C++ applications can link to the PVM library as well. There is a compatible set of primitives for Fortran as well, please see the PVM book for more information on these.

## 5.1 Process Control

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

```
int tid = pvm_mytid( void )
```

The routine `pvm_mytid()` returns the TID of this process and can be called multiple times. It enrolls this process into PVM if this is the first PVM call. Any PVM system call (not just `pvm_mytid`) will enroll a task in PVM if the task is not enrolled before the call, but it is common practice to call `pvm_mytid` first to perform the enrolling.

```
int info = pvm_exit( void )
```

The routine `pvm_exit()` tells the local `pvmd` that this process is leaving PVM. This routine does not kill the process, which can continue to perform tasks just like any other UNIX process. Users typically call `pvm_exit` right before exiting their C programs.

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

The routine `pvm_spawn()` starts up `ntask` copies of an executable file `task` on the virtual machine. `argv` is a pointer to an array of arguments to task with the end of the array specified by NULL. If task takes no arguments, then `argv` is NULL. The `flag` argument is used to specify options, and is a sum of:

```
Value    Option           Meaning
--------------------------------------------------------------------------
    0    PvmTaskDefault   PVM chooses where to spawn processes.
    1    PvmTaskHost      where argument is a particular host to spawn on.
    2    PvmTaskArch      where argument is a PVM_ARCH to spawn on.
    4    PvmTaskDebug     starts tasks under a debugger.
    8    PvmTaskTrace     trace data is generated.
   16    PvmMppFront      starts tasks on MPP front-end.
   32    PvmHostCompl     complements host set in where.
--------------------------------------------------------------------------
```

These names are predefined in `pvm3/include/pvm3.h`.

```
int info = pvm_kill( int tid )
```

The routine `pvm_kill()` kills some other PVM task identified by TID. This routine is not designed to kill the calling task, which should be accomplished by calling `pvm_exit()` followed by `exit()`.

```
int info = pvm_catchout( FILE *ff )
```

The default is to have PVM write the stderr and stdout of spawned tasks to the log file `/tmp/pvml.<uid>`. The routine `pvm_catchout` causes the calling task to catch output from tasks subsequently spawned. Characters printed on stdout or stderr in children tasks are collected by the `pvmds` and sent in control messages to the parent task, which tags each line and appends it to the specified file (in C). Each of the prints is prepended with information about which task generated the print, and the end of the print is marked to help separate outputs coming from several tasks at once.

If `pvm_exit` is called by the parent while output collection is in effect, it will block until all tasks sending it output have exited, in order to print all their output. To avoid this, one can turn off the output collection by calling `pvm_catchout(0)` before calling `pvm_exit`.

## 5.2 Dynamic Configuration

```
int info = pvm_addhosts( char **hosts, int nhost, int *infos)
int info = pvm_delhosts( char **hosts, int nhost, int *infos)
```

These routines add or delete a set of hosts in the virtual machine. Information is returned as the number of hosts successfully added. The argument `infos` is an array of length `nhost` that contains the status code for each individual host being added or deleted. This allows the user to check whether only one of a set of hosts caused a problem rather than trying to add or delete the entire set of hosts again.

These routines are sometimes used to set up a virtual machine, but more often they are used to increase the flexibility and fault tolerance of a large application. These routines allow an application to increase the available computing power (adding hosts) if it determines the problem is getting harder to solve.

## 5.3 Message Passing

The PVM communication model assumes that any task can send a message to any other 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 any particular limitations and assumes sufficient memory is available. The PVM communication model provides asynchronous blocking send, asynchronous blocking receive, and nonblocking receive functions. In their 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 nonblocking 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.

### Sending and Receiving Messages

Sending a message comprises three steps in PVM. First, a send buffer must be initialized by a call to `pvm_initsend()` or `pvm_mkbuf()`. Second, the message must be "packed" into this buffer using any number and combination of `pvm_pk*()` routines. Third, the completed message is sent to another process by calling the `pvm_send()` routine or multicast with the `pvm_mcast()` routine.

A message is received by calling either a blocking or nonblocking receive routine and then "unpacking" each of the packed items from the receive buffer. The receive routines can be set to accept any message, or any message from a specified source, or any message with a specified message tag, or only messages with a given message tag from a given source. There is also a probe function that returns whether a message has arrived, but does not actually receive it.

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

### 5.4 More Programming Help

For additional information about programming using PVM, I highly recommend reading the portions of the PVM book (available on the web page) dealing with programming, especially the sections on the PVM User Interface and Program Examples, from which much of this section was taken. There are also a number of examples available online, which you've already copied into your own `pvm3` directory, which can be very helpful.

## 6 Other PVM Details

In spawning tasks, PVM checks the load on each host in the virtual machine before deciding where to place a task. By default, PVM could decide to place two tasks on one lightly loaded host instead of putting one of the tasks on a heavily loaded host. If you wish to ensure that each host gets one and only one task, you need to force PVM to do this. See the options for the `pvm_spawn` routine for a method of doing this. (Take note of the `flag` and `where` parameters.) Of course, you may discover other methods for achieving this goal.

The method of sending messages between tasks is similar to that found on some supercomputers (e.g. you can label messages with tag IDs). However, there are a few differences:

- You need to initialize the message buffer (via `pvm_initsend`) before sending each message;
- You need to pack the data into a message before sending it;
- After receiving a message, you need to unpack the data.