ORNL/TM-12187

Engineering Physics and Mathematics Division

Mathematical Sciences Section

PVM 3 USER'S GUIDE AND REFERENCE MANUAL

Al Geist *
Adam Beguelin +
Jack Dongarra ***
Weicheng Jiang **
Robert Manchek **
Vaidy Sunderam ++
pvm@msr.epm.ornl.gov

* Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367
** University of Tennessee
Knoxville, TN 37996-1301
\+ Carnegie Mellon University and
Pittsburgh Supercomputing Center
Pittsburgh, PA 15213-3890
++ Emory University
Atlanta, GA 30322

Date Published: September, 1994

Research was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy, the National Science Foundation, and the State of Tennessee.

Prepared by the Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
operated by Martin Marietta Energy Systems, Inc.
for the U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

Contents

PVM 3 USER'S GUIDE
AND REFERENCE MANUAL

Al Geist
Adam Beguelin
Jack Dongarra
Weicheng Jiang
Robert Manchek
Vaidy Sunderam
pvm@msr.epm.ornl.gov

Abstract

This report is the PVM version 3.3 users' guide and reference manual. It contains an overview of PVM, and how version 3 can be obtained, installed and used.

PVM stands for Parallel Virtual Machine. It is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. PVM consists of two parts: a daemon process that any user can install on a machine, and a user library that contains routines for initiating processes on other machines, for communicating between processes, and changing the configuration of machines.

New features in this release are pointed out and described in detail. In addition this report describes the internal workings of version 3 and gives the user interface specifications. It describes several popular programming paradigms, which PVM supports, and gives several example programs in C and Fortran77. The report discusses issues and options regarding load balancing, performance, and fault tolerance. Basic steps for debugging PVM programs are presented, and references to additional PVM monitoring and visualization tools are given.

`What is new from last release of this User Guide. Fixed many typos, added more information about using PVM with the Intel Paragon, improved the troubleshooting startup section`

1. Introduction

This users' guide to PVM (Parallel Virtual Machine) version 3 contains examples and information needed for the straightforward use of PVM's basic features. Appendices contain full documentation of all PVM 3.3 options and error conditions as well as a quick reference.

PVM 3 is a software system that permits a network of heterogeneous UNIX computers to be used as a single large parallel computer. Thus large computational problems can be solved by using the aggregate power of many computers.

The development of PVM started in the summer of 1989 at Oak Ridge National Laboratory (ORNL) and is now an ongoing research project involving Vaidy Sunderam at Emory University, Al Geist at ORNL, Robert Manchek at the University of Tennessee (UT), Adam Beguelin at Carnegie Mellon University and Pittsburgh Supercomputer Center, Weicheng Jiang at UT, Jim Kohl, Phil Papadopoulos, June Donato, and Honbo Zhou at ORNL, and Jack Dongarra at ORNL and UT. It is a basic research effort aimed at advancing science, and is wholly funded by research appropriations from the U.S. Department of Energy, the National Science Foundation, and the State of Tennessee. Owing to its experimental nature, the PVM project produces, as incidental products, software that is of utility to researchers in the scientific community and to others. This software is, and has been distributed freely in the interest of advancement of science and is being used in computational applications around the world.

Under PVM, a user defined collection of serial, parallel, and vector computers appears as one large distributed-memory computer. Throughout this report the term *virtual machine* will be used to designate this logical distributed-memory computer, and *host* will be used to designate one of the member computers. PVM supplies the functions to automatically start up tasks on the virtual machine and allows the tasks to communicate and synchronize with each other. A *task* is defined as a unit of computation in PVM analogous to a UNIX process. It is often a UNIX process, but not necessarily so. Applications, which can be written in Fortran77 or C, can be parallelized by using message-passing constructs common to most distributed-memory computers. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel.

PVM supports heterogeneity at the application, machine, and network level. In other words, PVM allows application tasks to exploit the architecture best suited to their solution. PVM handles all data conversion that may be required if two computers use different integer or floating point representations. And PVM allows the virtual machine to be interconnected by a variety of different networks.

The PVM system is composed of two parts. The first part is a daemon, called *pvmd3* and sometimes abbreviated *pvmd*, that resides on all the computers making up the virtual machine. (An example of a daemon program is *sendmail* which handles all the incoming and outgoing electronic mail on a UNIX system.) Pvmd3 is designed so any user with a valid login can install this daemon on a machine. When a user wants to run a PVM application, he first creates a virtual machine by starting up PVM. The PVM application can then be started from a UNIX prompt on any of the hosts. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously.

The second part of the system is a library of PVM interface routines (`libpvm3.a`). This library contains user callable routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine. Application programs must be linked with this library to use PVM.

2. Features in PVM 3

PVM version 3 has many improvements over version 2 [1]. The following sections describe the features that are available in PVM 3.

2.1. Updated User interface

There are name conflicts between PVM 2.x routines and some multiprocessor libraries supplied by computer vendors. For example, the PVM 2.4 routine `barrier()` is also used (with slightly different functionality) on several multiprocessors. To avoid name conflicts all the PVM 3 user routines begin with `pvm_` in C and with `pvmf` in Fortran. We also incorporated new arguments and features into the interface to make it more flexible to application developers.

Although the user interface has been completely updated, conversion of PVM 2.4 applications to PVM 3 is straightforward. Appendix B contains a table of the mapping of routine names from PVM 2.4 to PVM 3. For users not wanting to convert their applications, PVM 2.4.2 will remain available from `netlib@ornl.gov`.

2.2. Integer Task Identifier

All processes that enroll in PVM 3 are represented by an integer task identifier. This is a change from version 2 of PVM which used a component name and instance number pair. Throughout this report the task identifier is represented by *tid*. The tid is the primary and most efficient method of identifying processes in PVM. Since tids must be unique across the entire virtual machine, they are supplied by the local pvmd and are not user chosen. PVM 3 contains several routines that return tid values so that the user application can identify other processes in the system. These routines are pvm_mytid(), pvm_spawn(), pvm_parent(), pvm_bufinfo(), pvm_tasks(), pvm_tidtohost(), and pvm_gettid().

Although less efficient, processes can still be identified by a name and instance number by joining a group. A user defines a group name and PVM returns a unique instance number for this process in this group.

2.3. Process Control

PVM supplies routines that enable a user process to become a PVM task and to become a normal process again. There are routines to add and delete hosts from the virtual machine, routines to start up and terminate PVM tasks, routines to send signals to other PVM tasks, and routines to find out information about the virtual machine configuration and active PVM tasks.

New capabilities in PVM 3.3 include the ability to register special PVM tasks to handle the jobs of adding new hosts, mapping tasks to hosts, and starting new tasks. This creates an interface for advanced batch schedulers (examples include Condor [3], DQS [2], and LSF [5]) to plug into PVM and run PVM jobs in batch mode. These register routines also allow debugger writers to plug into PVM and create sophisticated debuggers for PVM.

2.4. Fault Tolerance

If a host fails, PVM will automatically detect this and delete the host from the virtual machine. The status of hosts can be requested by the application, and if required a replacement host can be added by the application. It is still the responsibility of the application developer to make his application tolerant of host failure. PVM makes no attempt to automatically recover tasks that are killed because of a host failure. Another use of this feature would be to add more hosts as they become available, for example on a weekend, or if the application dynamically determines it could use more computational power.

2.5. Dynamic Process Groups

Dynamic process groups are implemented on top of PVM 3. In this implementation, a process can belong to multiple groups, and groups can change dynamically at any time during a computation.

Functions that logically deal with groups of tasks such as broadcast and barrier use the user's explicitly defined group names as arguments. Routines are provided for tasks to join and leave a named group. Tasks can also query for information about other group members.

2.6. Signaling

PVM provides two methods of signaling other PVM tasks. One method sends a UNIX signal to another task. The second method notifies a task about an event by sending it a message with a user-specified tag that the application can check for. Several notification events are available in PVM 3 including the exiting of a task, the deletion (or failure) of a host, and the addition of a host.

2.7. Communication

PVM provides routines for packing and sending messages between tasks. The 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 which 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. 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. So the maximum message size that can be sent or received is limited only by the amount of available memory on a given host.

2.8. Multiprocessor Integration

PVM was originally developed to join machines connected by a network into a single logical machine. Some of these hosts may themselves be parallel computers with multiple processors connected by a proprietary network or shared-memory.

With PVM 3 the dependence on UNIX sockets and TCP/IP software is relaxed. For example, programs written in PVM 3 can run on a network of SUN's, on a group of nodes on an Intel Paragon, on multiple Paragons connected by a network, or a heterogeneous combination of multiprocessor computers distributed around the world without having to write any vendor specific message-passing code. PVM 3 is designed to use native communication calls within a distributed memory multiprocessor or global memory within a shared memory multiprocessor. Messages between two nodes of a multiprocessor go directly between them while messages destined for a machine out on the network go to the user's single PVM daemon on the multiprocessor for further routing.

The Intel iPSC/860 and Paragon have been integrated into PVM 3 so that Intel's NX message-passing routines are used for inter-node communication. Thinking Machine Corporation's CM-5 has also been integrated using their CMMD message-passing routines. Cray and Convex supply their own optimized versions of PVM 3 for their T3D and Meta machines respectively. Other vendors including DEC, KSR, and IBM have also decided to supply PVM 3 with their respective multiprocessors.

PVM 3.3 includes shared memory ports to multiprocessor SPARCs, such as the SPARC-10, and the SGI Challenge series. More multiprocessor machines will be added to subsequent PVM 3 releases.

3. Getting and Installing PVM

PVM does not require special privileges to be installed. Anyone with a valid login on the hosts can do so. Only one person at an organization needs to get and install PVM for everyone at that organization to use it. *PVM_ARCH* is used throughout this report to represent the architecture name PVM uses for a given computer. Table 1 lists all the PVM_ARCH names and their corresponding architecture types that are supported in PVM 3.3.

3.1. Obtaining PVM

There are several ways to obtain the software and documentation. This user's guide, the PVM 3 source code, man pages, XPVM, and pointers to other PVM related packages are available on *netlib*. Netlib is a software distribution service set up on the Internet There are several ways to get software from netlib. The first is with a tool called *xnetlib*. 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@ornl.gov` with the message `send xnetlib.shar from xnetlib` or anonymous ftp from `cs.utk.edu pub/xnetlib`.

Netlib files can also be obtained by anonymous ftp to `netlib2.cs.utk.edu`. Look in directory pvm3. The file `index` describes the files in this directory.

The PVM software can be requested by email. To receive this software send email to `netlib@ornl.gov` with the message: `send index from pvm3`. An automatic mail handler will return a list of available files and further instructions by email. The advantage of this method is that anyone with email access to Internet can obtain the software.

3.2. Unpacking

The source files, which consume about 1 Mbyte when unpacked, are available in uuencoded/compressed *tar* format. Place the file in the directory where you want to install the source. By default PVM assumes it is installed in your $HOME/pvm3 , but it can be installed in a more centralized area like `/usr/local/pvm3`. To unpack the source:

```
% uudecode pvm3.3.0.tar.z.uu
% uncompress pvm3.3.0.tar.Z
% tar xvf pvm3.3.0.tar
```

3.3. Building

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

The easiest method is to set these two variables in your `.cshrc` file. Here is an example for PVM_ROOT:

PVM_ARCH	Machine	Notes
AFX8	Alliant FX/8	
ALPHA	DEC Alpha	DEC OSF-1
BAL	Sequent Balance	DYNIX
BFLY	BBN Butterfly TC2000	
BSD386	80386/486 Unix box	BSDI
CM2	Thinking Machines CM2	Sun front-end
CM5	Thinking Machines CM5	
CNVX	Convex C-series	
CNVXN	Convex C-series	native mode
CRAY	C-90, YMP	UNICOS
CRAY2	Cray-2	
CRAYSMP	Cray S-MP	
DGAV	Data General Aviion	
E88K	Encore 88000	
HP300	HP-9000 model 300	HPUX
HPPA	HP-9000 PA-RISC	
I860	Intel iPSC/860	link -lrpc
IPSC2	Intel iPSC/2 386 host	SysV
KSR1	Kendall Square KSR-1	OSF-1
LINUX	80386/486 LINUX box	LINUX
MASPAR	MASPAR host	
MIPS	MIPS 4680	
NEXT	NeXT	
PGON	Intel Paragon	link -lrpc
PMAX	DECstation 3100, 5100	Ultrix
RS6K	IBM/RS6000	AIX
RT	IBM RT	
SGI	Silicon Graphics	IRIX 4.x
SGI5	Silicon Graphics	IRIX 5.1
SUN3	Sun 3	SunOS 4.2
SUN4	Sun 4, SPARCstation	SunOS 4.2
SUN4SOL2	Sun 4, SPARCstation	Solaris 2.2
SYMM	Sequent Symmetry	
U370	IBM 370	AIX
UVAX	DEC MicroVAX	

Table 1: PVM_ARCH names used in PVM 3.

```
setenv PVM_ROOT /home/msr/u2/kohl/pvm3
```

The recommended method to set PVM_ARCH is to append the file `PVM_ROOT/lib/cshrc.stub` onto your `.cshrc` file 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.

The PVM source comes with directories and makefiles for most machines you are likely to have. Building for each architecture type is done automatically by going into the PVM_ROOT directory and typing `make`. The makefile will automatically determine which architecture it is being executed on and build `pvmd3`, `libpvm3.a`, `libfpvm3.a`, `pvmgs` and `libgpvm3.a`. It places all these files in `pvm3/lib/PVM_ARCH` with the exception of pvmgs which is placed in `PVM_ROOT/bin/PVM_ARCH`..

To build PVM for the Intel Paragon or iPSC/860 the above instructions work if you are on these machines. Note that a node specific version of `libpvm3.a` will also be built as `libpvm3pe.a`. The iPSC/860 will also create a node specific version of `libfpvm3.a` called libfpvm3pe.a because the host and nodes use different CPUs. If you are on a SUN or SGI with Intel cross compilers, then you will need to type `make PVM_ARCH=PGON` or `make PVM_ARCH=CUBE` respectively for the Paragon and iPSC/860. See the file pvm3/Readme.mpp for the latest MPP building instructions on all supported machines.

3.4. Installing

PVM looks for user executables in the default location `$HOME/pvm3/bin/PVM_ARCH`. If PVM is installed in a single location like `/usr/local` for all users, then each user should still create `$HOME/pvm3/bin/PVM_ARCH` to place his own executables. For example, if a user's PVM application wants to spawn a task called *foo* on a SPARCstation called sunny, then on sunny there should be an executable file `$HOME/pvm3/bin/SUN4/foo`. This default can be changed to a different search path in the hostfile.

4. PVM Console

The PVM console, called `pvm`, is a stand alone PVM task which allows the user to interactively start, query and modify the virtual machine. The console may be started and stopped multiple times on any of the hosts in the virtual machine without affecting PVM or any applications that may be running.

When started, `pvm` determines if PVM is already running and if not automatically executes pvmd on this host, passing pvmd the command line options and hostfile. Thus PVM need not be running to start the console.

```
pvm [-d<debugmask>] [hostfile]
pvm -n<hostname>
```

Debugmask is a hex number corresponding to the debug bits from pvmd.c See the "Implementation" section for more details on the debugmask.

The -n option is useful for specifying an alternate name for the master pvmd ((in case hostname doesn't match the IP address you want). This is useful if a host has a multiple networks connected to it such as FDDI or ATM, and you want PVM to use a particular network.

Once started the console prints the prompt:

```
pvm>
```

and accepts commands from standard input. If you get the message "Can't Start pvmd", then check the Troubleshooting Startup section and try again.

The available console commands are:

add followed by one or more host names will add these hosts to the virtual machine.

alias define or list 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. 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 which can be used to get information about any of the interactive commands. Help may be followed by a command name which will list options and flags available for this command.

id print console task id.

jobs list running jobs.

kill can be used to terminate any PVM process,

mstat show status of specified hosts.

ps -a lists all processes currently on the virtual machine, their locations, their task IDs, and their parents' task IDs.

pstat show status of a single PVM process.

quit exit 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 display or set environment variables.

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

spawn start a PVM application. Options include:

- **-count** number of tasks, default is 1.
- **-(host)** spawn on host, default is any.
- **-(PVM_ARCH)** spawn of hosts of type PVM_ARCH.
- **-?** enable debugging.
- -> redirect task output to console.
- ->**file** redirect task output to file.
- ->>**file** redirect task output append to file.

unalias undefine command alias.

version print version of libpvm being used.

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

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

The two most popular methods of running PVM 3 are to start `pvm` then add hosts manually (`pvm` also accepts an optional hostfile argument) or to start `pvmd3` with a hostfile then start `pvm` if desired.

To shut down PVM type `halt` at a PVM console prompt.

4.1. Host File Options

The *hostfile* defines the initial configuration of hosts that PVM combines into a virtual machine. It also contains information about hosts that the user may wish to add to the configuration later.

Only one person at a site needs to install PVM, but each PVM user should have their own hostfile, which describes their own personal virtual machine.

The hostfile in its simplest form is just a list of hostnames one to a line. Blank lines are ignored, and lines that begin with a # are comment lines. This allows the user to document his hostfile and also provides a handy way to modify the initial configuration by commenting out various hostnames (see Figure 1).

Several options can be specified on each line after the hostname. The options are separated by white space.

lo= userid allows the user to specify an alternate login name for this host; otherwise, his login name on the start-up machine is used.

```
# configuration used for my run
sparky
azure.epm.ornl.gov
thud.cs.utk.edu
sun4
```

Figure 1: Simple hostfile lists virtual machine configuration.

so=pw will cause PVM to prompt the user for a password on this host. This is useful in the cases where the user has a different userid and password on a remote system. PVM uses rsh by default to start up remote pvmd's, but when pw is specified PVM will use rexec() instead.

dx= location_of_pvmd This allows the user to specify a location other than the default for this host. This is useful if someone wants to use his own personal copy of pvmd,

ep= paths_to_user_executables This allows the user to specify a series of paths to search down to find the requested files to spawn on this host. Multiple paths are separated by a colon. If `ep=` is not specified, then PVM looks for the application tasks in `$HOME/pvm3/bin/PVM_ARCH`.

sp= value Specifies the relative computational speed of the host compared to other hosts in the configuration. The range of possible values is 1 to 1000000 with 1000 as the default.

bx= location_of_debugger Specifies which debugger script to invoke on this host if debugging is requested in the spawn routine. Note: the environment variable PVM_DEBUGGER can also be set. The default debugger is `pvm3/lib/debugger`.

wd= working_directory Specifies a working directory in which all spawned tasks on this host will execute. The default is `$HOME`.

so=ms Specifies that user will manually start a slave pvmd on this host. Useful if rsh and rexec network services are disabled but IP connectivity exists. When using this option you will see in the tty of the pvmd3:

```
[t80040000] ready   Fri Aug 27 18:47:47 1993
*** Manual startup ***
Login to "honk" and type:
pvm3/lib/pvmd -S -d0 -nhonk 1 80a9ca95:0cb6 4096 2 80a95c43:0000
Type response:
```

on honk after typing the given line, you should see:

```
ddpro<2312> arch<ALPHA> ip<80a95c43:0a8e> mtu<4096>
```

which you should relay back to the master pvmd. At that point, you will see:

```
Thanks
```

and the two pvmds should be able to communicate.

If the user wants to set any of the above options as defaults for a series of hosts, then the user can place these options on a single line with a * for the hostname field. The defaults will be in effect for all the following hosts until they are overridden by another set-defaults line.

Hosts that the user doesn't want in the initial configuration but may add later can be specified in the hostfile by beginning those lines with an &. An example hostfile displaying most of these options is shown in Figure 2.

```
# Comment lines start with # (blank lines ignored)
gstws
ipsc dx=/usr/geist/pvm3/lib/I860/pvmd3
ibm1.scri.fsu.edu lo=gst so=pw

# set default options for following hosts with *
* ep=$sun/problem1:~/nla/mathlib
sparky
#azure.epm.ornl.gov
midnight.epm.ornl.gov

# replace default options with new values
* lo=gageist so=pw ep=problem1
thud.cs.utk.edu
speedy.cs.utk.edu

# machines for adding later are specified with &
# these only need listing if options are required
&sun4   ep=problem1
&castor dx=/usr/local/bin/pvmd3
&dasher.cs.utk.edu lo=gageist
&elvis  dx=~/pvm3/lib/SUN4/pvmd3
```

Figure 2: PVM hostfile illustrating all options.

4.2. Troubleshooting Startup

If PVM has a problem starting up, it will print an error message either to the screen or in the log file `/tmp/pvml.<uid>`. This section should help in interpreting the error message and explain how to solve the problem.

If the message says

```
[t80040000] Can't start pvmd
```

First check that your `.rhosts` file on the remote host contains the name of the host from which you are starting PVM. An external check that your `.rhosts` file is set correctly is to type:

```
% rsh remote_host 'ls'
```

Other reasons to get this message include not having PVM installed on a host or not having PVM_ROOT set correctly on some host. You can check this by typing:

```
% rsh remote_host 'printenv'
```

Some Unix shells, for example ksh, do not set environment variables on remote hosts when using rsh. In PVM 3.3 there are two work arounds for such shells. First, if you set the environment variable, PVM_DPATH, on the master host to `pvm3/lib/pvmd`, then this will override the default dx path. The second method is to tell PVM explicitly were to find the remote pvmd executable by using the `dx=` option in the hostfile.

If PVM is manually killed, or stopped abnormally, (an example is system crash) then check for the existence of the file `/tmp/pvmd.<uid>`. This file is used for authentication and should exist only while PVM is running. If this file is left behind, it prevents PVM from starting. Simply delete this file.

If the message says

```
[t80040000] Login incorrect
```

then it probably means that there is not an account on the remote machine with your login name. If your login name is different on the remote machine, the fix is to use the `lo=` option in the hostfile.

If you get any other strange messages, then check your `.cshrc` file. It is important that the user not have any I/O in his `.cshrc` file because this will interfere with the start up of PVM. If you want to print out stuff when you log in, such as who or uptime, you should either do it in your `.login` script or put the commands in an "if" statement to ensure that stuff only gets printed when you are logging in interactively, not when you're running a csh command script. Here is an example of how this can be done:

```
if ( { tty -s } && $?prompt )  then
    echo terminal type is $TERM
    stty erase '^?' kill '^u' intr '^c' echo
endif
```

4.3. Compiling PVM Applications

A C program that makes PVM calls needs to be linked with `libpvm3.a`. If the program also makes use of dynamic groups, then it should be linked to libgpvm3.a before `libpvm3.a`. A Fortran program using PVM needs to be linked with libfpvm3.a and `libpvm3.a`. And if it uses dynamic groups then it needs to be linked to libfpvm3.a, libgpvm3.a, and `libpvm3.a` in that order.

PVM programs that are being compiled to run on the nodes of an Intel i860 should be linked to `libpvm3pe.a` and `libfpvm3pe.a` instead of `libpvm3.a` and libfpvm3.a.

On the Intel Paragon, PVM programs can run on either the service or compute nodes. Programs that are being compiled to run on the compute nodes should be linked to libpvm3pe.a, while programs designed to run on service nodes should be linked to libpvm3.a. Master/slave applications, where the master runs on a service node and the slaves run on compute nodes, would thus require different library specifications in a Makefile. FORTRAN programs should link to libfpvm3.a first and then either libpvm3pe.a or libpvm3.a. All PVM applications on the Paragon also require linking with NXLIB and the Remote Procedure Call (rpc) libraries, as PVM requires them. Applications compiled with either GNU C (gcc) or C++ also require the Mach libraries.

The following table summarizes which libraries must be linked on the Paragon:

```
Application              Application written in:
  Runs on:                  C          FORTRAN
-------------------------------------------------
  Service             libpvm3.a      libfpvm3.a
  Partition           -lrpc          libpvm3.a
                      -lnx           -lrpc
                      -lmach (*)     -lnx
-------------------------------------------------
  Compute             libpvm3pe.a    libfpvm3.a
  Partition           -lrpc          libpvm3pe.a
                      -lnx           -lrpc
                      -lmach (*)     -lnx
-------------------------------------------------

  (*) must also be included for GNU C or C++
```

The order of the libraries (from top to bottom for a given case) is important. The example makefile for the Paragon in the PVM_ROOT/examples/PGON directory provides a working example of the proper library links. A program compiled for the service partition will not run in the compute partition, and vice versa; in both instances the application will either hang or fail to perform message passing properly.

For all machines, example programs and makefile are supplied with the PVM source code in the directory `pvm3/examples`. A `Readme` file in this directory describes how to build and run the examples. The makefile demonstrates how C and Fortran applications should be linked with the PVM libraries. The makefile also contains information in its header about additional libraries required on some architectures. An "architecture independent" make program is supplied with PVM. This script is located in pvm3/lib/aimk and automatically detects what kind of architecture it is running on and adds the correct additional libraries. To build any of the examples you can type:

```
% aimk example_name
```

4.4. Running PVM Applications

Once PVM is running, an application using PVM routines can be started from a UNIX command prompt on any of the hosts in the virtual machine. An application need not be started on the same machine the user happens to start PVM.

Stdout and stderr appear on the screen for all manually started PVM tasks. The standard error from spawned tasks is written to the log file `/tmp/pvml.<uid>` on the host where PVM was started. The easiest way to see standard output from spawned PVM tasks is to use the redirection available in the pvm console. If standard output is not redirected at the pvm console, then this output also goes to the log file.

Users sometimes want to run their programs with a *nice* value that is at a lower priority so the programs impinge less on workstation owners. There are a couple of ways to accomplish this. The first method, which works with both Fortran and C applications, is to replace your program with a shell script that starts your program. Here is an example two line script:

```
#!/bin/sh
exec nice -10 your_program $*
```

Then when you spawn the shell script it will exec your program at a *nice* level. The second method is to call the UNIX function setpriority() in your program.

A whole series of applications may be run on the existing PVM configuration. It is not necessary to start a new PVM for each application, although it may be necessary to reset PVM if an application crashes.

It is also possible to compile PVM with -DOVERLOADHOST defined. This allows a user to create overlapping virtual machines. The next sections will describe how to write PVM application programs.

5. User Interface

An alphabetical listing of all the routines is given in Appendix A. Appendix A contains a detailed description of each routine, including a description of each argument in each routine and the possible error codes a routine may return and the possible reasons for the error. Each listing includes examples of both C and Fortran use.

A concise summary of the PVM 3.3 routines can be found on the PVM quick reference guide.

In this section we give a brief description of the routines in the PVM 3.3 user library. This section is organized by the functions of the routines. For example, in the subsection on *Dynamic Configuration* is a discussion of the purpose of dynamic configuration, how a user might take advantage of this functionality, and the C and Fortran PVM routines that pertain to this function.

In PVM 3 all PVM tasks are identified by an integer supplied by the local pvmd. In the following descriptions this identifier is called tid. It is similar to the process ID (PID) used in the UNIX system except the tid has encoded in it the location of the process in the virtual machine. This encoding allows for more efficient communication routing, and allows for more efficient integration into multiprocessors.

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

5.1. Process Control

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

```
call pvmfmytid( tid )
```

The routine pvm_mytid() enrolls this process into PVM on its first call and generates a unique `tid` if the process was not started with pvm_spawn(). It returns the `tid` of this process and can be called multiple times. Any PVM system call (not just pvm_mytid) will enroll a task in PVM if the task is not enrolled before the call.

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

```
call pvmfexit( info )
```

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.

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

```
call pvmfspawn(  task, flag, where, ntask, tids, numt )
```

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

PvmTaskDefault - PVM chooses where to spawn processes,

PvmTaskHost - the `where` argument specifies a particular host to spawn on,

PvmTaskArch - the `where` argument specifies a PVM_ARCH to spawn on,

PvmTaskDebug - starts these processes up under debugger,

PvmTaskTrace - the PVM calls in these processes will generate trace data.

PvmMppFront - starts process up on MPP front-end/service node.

PvmHostCompl - starts process up on complement host set.

PvmTaskTrace is a new feature in PVM 3.3. To display the events, a graphical interface, called XPVM has been created. XPVM combines the features of the PVM console, the Xab debugging package, and ParaGraph to display real-time or post mortem executions. XPVM is available on netlib.

On return `numt` is set to the number of tasks successfully spawned or an error code if no tasks could be started. If tasks were started, then pvm_spawn() returns a vector of the spawned tasks' `tids` and if some tasks could not be started the corresponding error codes are placed in the last $(ntask - numt)$ positions of the vector.

pvm_spawn() can also start tasks on multiprocessors. In the case of the Intel iPSC/860 the following restrictions apply. Each spawn call gets a subcube of size `ntask` and loads the program `task` on all of these nodes. The iPSC/860 OS has an allocation limit of 10 subcubes across all users, so it is better to start a block of tasks on an iPSC/860 with a single pvm_spawn() call rather than several calls. Two different blocks of tasks spawned separately on the iPSC/860 can still communicate with each other as well as any other PVM tasks even though they are in separate subcubes. The iPSC/860 OS has a restriction that messages going from the nodes to the outside world be less than 256 Kbytes.

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

```
call pvmfkill( tid, info )
```

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().

5.2. Information

```
int tid = pvm_parent( void )
```

```
call pvmfparent( tid )
```

The routine pvm_parent() returns the `tid` of the process that spawned this task or the value of PvmNoParent if not created by pvm_spawn().

```
int pstat = pvm_pstat( int tid )
```

```
call pvmfpstat( tid, pstat )
```

The routine pvm_pstat() returns the status of a PVM task identified by `tid`. It returns PvmOk if the task is running, PvmNoTask if not, or PvmBadParam if `tid` is invalid.

```
int mstat = pvm_mstat( char *host )
```

```
call pvmfmstat( host, mstat )
```

The routine pvm_mstat() returns PvmOk if `host` is running, PvmHostFail if unreachable, or PvmNoHost if `host` is not in the virtual machine. This information can be useful when implementing application level fault tolerance.

```
int info = pvm_config(  int *nhost, int *narch,
                        struct pvmhostinfo **hostp )
```

```
call pvmfconfig( nhost, narch, dtid, name, arch, speed, info )
```

The routine pvm_config() returns information about the virtual machine including the number of hosts, `nhost`, and the number of different data formats, `narch`. `hostp` is a pointer to an array of `pvmhostinfo` structures. The array is of size `nhost`. Each `pvmhostinfo` structure contains the pvmd tid, host name, name of the architecture, and relative cpu speed for that host in the configuration. PVM does not use or determine the speed value. The user can set this value in the hostfile and retrieve it with pvm_config() to use in an application. The Fortran function returns information about one host per call and cycles through all the hosts. Thus, if pvmfconfig is called nhost times, the entire virtual machine will be represented. The Fortran function does not reset itself until the end of a cycle. If the virtual machine is changing rapidly, pvmfconfig will not report the change until it is reset. The user can manually reset pvmfconfig in the middle of a cycle by calling pvmfconfig with `nhost = -1`.

```
int info = pvm_tasks(  int which, int *ntask,
                       struct pvmtaskinfo **taskp )
```

```
call pvmftasks( which, ntask, tid, ptid, dtid, flag, aout, info )
```

The routine pvm_tasks() returns information about the PVM tasks running on the virtual machine. The integer `which` specifies which tasks to return information about. The present options are (0), which means all tasks, a pvmd tid, which means tasks running on that host, or a tid, which means just the given task.

The number of tasks is returned in `ntask`. `taskp` is a pointer to an array of `pvmtaskinfo` structures. The array is of size `ntask`. Each `taskinfo` structure contains the tid, pvmd tid, parent tid, a status flag, and the spawned file name. (PVM doesn't know the file name of manually started tasks.) The Fortran function returns information about one task per call and cycles through all the tasks. Thus, if where = 0, and pvmftasks is called ntask times, all tasks will be represented. The Fortran function does not reset itself until the end of a cycle. If the number of tasks is changing rapidly, pvmftasks will not report the change until it is reset. The user can manually reset pvmftasks in the middle of a cycle by calling pvmftasks with `ntask = -1`.

```
int dtid = pvm_tidtohost( int tid )
```

```
call pvmftidtohost( tid, dtid )
```

If all a user needs to know is what host a task is running on, then pvm_tidtohost() can return this information.

5.3. Dynamic Configuration

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

```
call pvmfaddhost( host, info )
call pvmfdelhost( host, info )
```

The C routines add or delete a set of `hosts` in the virtual machine. The Fortran routines add or delete a single `host` in the virtual machine. In the Fortran routine `info` is returned as 1 or a status code. In the C version `info` 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 if only one of a set of hosts caused a problem rather than trying to add or delete the entire set of hosts again.

5.4. Signaling

```
int info = pvm_sendsig( int tid, int signum )
```

```
call pvmfsendsig( tid, signum, info )
```

pvm_sendsig() sends a signal `signum` to another PVM task identified by `tid`.

```
int info = pvm_notify( int what, int msgtag, int cnt, int tids )
```

```
call pvmfnotify( what, msgtag, cnt, tids, info )
```

The routine **`pvm_notify`** requests PVM to notify the caller on detecting certain events. The present options are:

PvmTaskExit - notify if a task exits.

PvmHostDelete - notify if a host is deleted (or fails).

PvmHostAdd - notify if a host is added.

In response to a notify request, some number of messages (see Appendix A) are sent by PVM back to the calling task. The messages are tagged with the code (msgtag) supplied to notify. The tids array specifies who to monitor when using TaskExit or HostDelete. The array contains nothing when using HostAdd. Outstanding notifies are consumed by each notification. For example, a HostAdd notification will need to be followed by another call to pvm_notify() if this task is to be notified of further hosts being added. If required, the routines pvm_config and pvm_tasks can be used to obtain task and pvmd tids.

If the host on which task A is running fails, and task B has asked to be notified if task A exits, then task B will be notified even though the exit was caused indirectly.

5.5. Setting and Getting Options

```
int oldval = pvm_setopt( int what, int val )
int val = pvm_getopt( int what )
```

```
call pvmfsetopt( what, val, oldval )
call pvmfgetopt( what, val )
```

The routines **`pvm_setopt`** and **`pvm_getopt`** are a general purpose function to allow the user to set or get options in the PVM system. In PVM 3 pvm_setopt can be used to set several options including: automatic error message printing, debugging level, and communication routing method for all subsequent PVM calls. pvm_setopt returns the previous value of set in oldval. The PVM 3.3 `what` can take have the following values:

Option value		MEANING
`PvmRoute`	1	routing policy
`PvmDebugMask`	2	debugmask
`PvmAutoErr`	3	auto error reporting
`PvmOutputTid`	4	stdout device for children
`PvmOutputCode`	5	output msgtag
`PvmTraceTid`	6	trace device for children
`PvmTraceCode`	7	trace msgtag
`PvmFragSize`	8	message fragment size
`PvmResvTids`	9	allow messages to be sent to reserved tags and tids

See Appendix A for allowable values for these options. Future expansions to this list are planned.

pvm_setopt() can set several communication options inside of PVM such as routing method or fragment sizes to use. It can be called multiple times during an application to selectively set up direct task-to-task communication links, but typical use is to call it once after the initial call to pvm_mytid(). For example:

```
CALL PVMFSETOPT( PvmRoute, PvmRouteDirect )
```

The advantage of direct links is the observed factor of two boost in communication performance. The drawback is the small number of direct links allowed by some UNIX systems, which makes their use unscalable.

When large messages are being sent over FDDI or HiPPI networks, communication performance can sometimes be improved by setting a large fragment size such as 64K.

5.6. Message Passing

Sending a message is composed of three steps in PVM. First, a send buffer must be initialized by a call to pvm_initsend() or pvm_mkbuf(). Second, the message must be "packed" into this buffer using any number and combination of pvm_pk*() routines. (In Fortran all message packing is done with the pvmfpack() subroutine.) Third, the completed message is sent to another process by calling the pvm_send() routine or multicast with the pvm_mcast() routine. In addition there are collective communication functions that operate over an entire group of tasks, for example, broadcast and scatter/gather.

PVM also supplies the routine, pvm_psend(), which combines the three steps into a single call. This allows for the possibility of faster internal implementations, particularly by MPP vendors. pvm_psend() only packs and sends a contiguous array of a single data type. pvm_psend() uses its own send buffer and thus doesn't affect a partially packed buffer to be used by pvm_send().

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

PVM also supplies the routine, pvm_precv(), which combines a blocking receive and unpack call. Like pvm_psend(), pvm_precv() is restricted to a contiguous array of a single data type. Between tasks running on an MPP such as the Paragon or T3D the user should receive a pvm_psend() with a pvm_precv(). This restriction was done because much faster MPP implementations are possible when pvm_psend() and pvm_precv() are matched. The restriction is only required within a MPP. When communication is between hosts, pvm_precv() can receive messages sent with pvm_psend(), pvm_send(), pvm_mcast(), or pvm_bcast(). Conversely, pvm_psend() can be received by any of the PVM receive routines.

If required, more general 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.

5.6.1. Message Buffers

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

```
int bufid = pvm_mkbuf( int encoding )
```

```
call pvmfmkbuf( encoding, bufid )
```

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

The `encoding` options are:

PvmDataDefault - XDR encoding is used by default because PVM can not know if the user is going to add a heterogeneous machine before this message is sent. If the user knows that the next message will only be sent to a machine that understands the native format, then he can use *PvmDataRaw* encoding and save on encoding costs.

PvmDataRaw - no encoding is done. Messages are sent in their original format. If the receiving process can not read this format, then it will return an error during unpacking.

PvmDataInPlace - data left in place. Buffer only contains sizes and pointers to the items to be sent. When pvm_send() is called the items are copied directly out of the user's memory. This option decreases the number of times the message is copied at the expense of requiring the user to not modify the items between the time they are packed and the time they are sent. Another use of this option would be to call pack once and modify and send certain items (arrays) multiple times during an application. An example would be passing of boundary regions in a discretized PDE implementation.

```
int bufid = pvm_initsend( int encoding )
```

```
call pvmfinitsend( encoding, bufid )
```

The routine pvm_initsend clears the send buffer and creates a new one for packing a new message. The encoding scheme used for this packing is set by `encoding`. The new buffer identifier is returned in `bufid`. If the user is only using a single send buffer then pvm_initsend() must be called before packing a new message into the buffer, otherwise the existing message will be appended.

```
int info = pvm_freebuf( int bufid )
```

```
call pvmffreebuf( bufid, info )
```

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

```
int bufid = pvm_getsbuf( void )
```

```
call pvmfgetsbuf( bufid )
```

pvm_getsbuf() returns the active send buffer identifier.

```
int bufid = pvm_getrbuf( void )
```

```
call pvmfgetrbuf( bufid )
```

pvm_getrbuf() returns the active receive buffer identifier.

```
int oldbuf = pvm_setsbuf( int bufid )
```

```
call pvmfsetsbuf( bufid, oldbuf )
```

This routine sets the active send buffer to `bufid`, saves the state of the previous buffer, and returns the previous active buffer identifier `oldbuf`.

```
int oldbuf = pvm_setrbuf( int bufid )
```

```
call pvmfsetrbuf( bufid, oldbuf )
```

This routine sets the active receive buffer to `bufid`, saves the state of the previous buffer, and returns the previous active buffer identifier `oldbuf`.

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

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

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

5.6.2. Packing Data

Each of the following C routines packs an array of the given data type into the active send buffer. They can be called multiple times to pack a single message. Thus a message can contain several arrays each with a different data type. There is no limit to the complexity of the packed messages, but an application should unpack the messages exactly like they were packed. C structures must be passed by packing their individual elements.

The arguments for each of the routines are a pointer to the first item to be packed, `nitem` which is the total number of items to pack from this array, and `stride` which is the stride to use when packing. An exception is pvm_pkstr() which by definition packs a NULL terminated character string and thus does not need `nitem` or `stride` arguments.

```
int info = pvm_pkbyte(   char   *cp, int nitem, int stride )
int info = pvm_pkcplx(   float  *xp, int nitem, int stride )
int info = pvm_pkdcplx(  double *zp, int nitem, int stride )
int info = pvm_pkdouble( double *dp, int nitem, int stride )
int info = pvm_pkfloat(  float  *fp, int nitem, int stride )
int info = pvm_pkint(    int    *np, int nitem, int stride )
int info = pvm_pklong(   long   *np, int nitem, int stride )
int info = pvm_pkshort(  short  *np, int nitem, int stride )
int info = pvm_pkuint(  unsigned int   *np, int nitem, int stride )
int info = pvm_pkushort(unsigned short *np, int nitem, int stride )
int info = pvm_pkulong( unsigned long  *np, int nitem, int stride )
int info = pvm_pkstr(    char   *cp )
```

```
int info = pvm_packf( const char *fmt, ... )
```

PVM also supplies a packing routine pvm_packf() that uses a printf-like format expression to specify what and how to pack data into the send buffer. All variables are passed as addresses if count and stride are specified; otherwise, variables are assumed to be values. A description of the format syntax is given in Appendix A.

A single Fortran subroutine handles all the packing functions of the above C routines.

```
call pvmfpack( what, xp, nitem, stride, info )
```

The argument `xp` is the first item of the array to be packed. Note that in Fortran the number of characters in a string to be packed must be specified in `nitem`. The integer `what` specifies the type of data to be packed. The supported options are:

STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

These names have been predefined in parameter statements in the include file `pvm3/include/fpvm3.h`. Some vendors may extend this list to include 64 bit architectures in their PVM implementations. We will be adding INTEGER8, REAL16, etc. as soon as XDR support for these data types is available.

5.6.3. Sending and Receiving Data

```
int info = pvm_send( int tid, int msgtag )
```

```
call pvmfsend( tid, msgtag, info )
```

The routine pvm_send() labels the message with an integer identifier `msgtag` and sends it immediately to the process `tid`.

```
int info = pvm_mcast( int *tids, int ntask, int msgtag )
```

```
call pvmfmcast( ntask, tids, msgtag, info )
```

The routine pvm_mcast() labels the message with an integer identifier `msgtag` and broadcasts the message to all tasks specified in the integer array `tids` (except itself). The `tids` array is of length `ntask`.

```
int info = pvm_psend( int tid, int msgtag, void *vp, int cnt, int type )
```

```
call pvmfpsend( tid, msgtag, xp, cnt, type, info )
```

The routine pvm_psend() packs and sends an array of the specified datatype to the task identified by `tid`. The defined datatypes for Fortran are the same as for pvmfpack(). In C the `type` argument can be any of the following:

PVM_STR	PVM_FLOAT
PVM_BYTE	PVM_CPLX
PVM_SHORT	PVM_DOUBLE
PVM_INT	PVM_DCPLX
PVM_LONG	PVM_DCPLX
PVM_USHORT	PVM_UINT
PVM_ULONG	

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

```
int bufid = pvm_recv( int tid, int msgtag )
```

```
call pvmfrecv( tid, msgtag, bufid )
```

This blocking receive routine will wait until a message with label `msgtag` has arrived from `tid`. A value of -1 in `msgtag` or `tid` matches anything (wildcard). It then places the message in a new active receive buffer that is created. The previous active receive buffer is cleared unless it has been saved with a pvm_setrbuf() call.

```
int bufid = pvm_nrecv( int tid, int msgtag )
```

```
call pvmfnrecv( tid, msgtag, bufid )
```

If the requested message has not arrived, then the non-blocking receive pvm_nrecv() returns `bufid` = 0. This routine can be called multiple times for the same message to check if it has arrived while performing useful work between calls. When no more useful work can be performed the blocking receive pvm_recv() can be called for the same message. If a message with label `msgtag` has arrived from `tid`, pvm_nrecv() places this message in a new active receive buffer which it creates and returns the ID of this buffer. The previous active receive buffer is cleared unless it has been saved with a pvm_setrbuf() call. A value of -1 in `msgtag` or `tid` matches anything (wildcard).

```
int bufid = pvm_probe( int tid, int msgtag )
```

```
call pvmfprobe( tid, msgtag, bufid )
```

If the requested message has not arrived, then pvm_probe() returns `bufid` = 0. Otherwise, it returns a bufid for the message, but does not "receive" it. This routine can be called multiple times for the same message to check if it has arrived while performing useful work between calls. In addition pvm_bufinfo() can be called with the returned bufid to determine information about the message before receiving it.

```
int info = pvm_bufinfo( int bufid, int *bytes, int *msgtag, int *tid )
```

```
call pvmfbufinfo( bufid, bytes, msgtag, tid, info )
```

```
int bufid = pvm_trecv( int tid, int msgtag, struct timeval *tmout )
```

```
call pvmftrecv( tid, msgtag, sec, usec, bufid )
```

PVM also supplies a timeout version of receive. Consider the case where a message is never going to arrive (due to error or failure). The routine pvm_recv would block forever. There are times when the user wants to give up after waiting for a fixed amount of time. The routine pvm_trecv() allows the user to specify a timeout period. If the timeout period is set very large then pvm_trecv acts like pvm_recv. If the timeout period is set to zero then pvm_trecv acts like pvm_nrecv. Thus, pvm_trecv fills the gap between the blocking and nonblocking receive functions.

The routine pvm_bufinfo() returns msgtag, source tid, and length in bytes of the message identified by `bufid`. It can be used to determine the label and source of a message that was received with wildcards specified.

```
int info = pvm_precv(  int tid, int msgtag, void *vp, int cnt,
                       int type, int *rtid, int *rtag, int *rcnt )
```

```
call pvmfprecv( tid, msgtag, xp, cnt, type, rtid, rtag, rcnt, info )
```

The routine pvm_precv() combines the functions of a blocking receive and unpacking the received buffer. It does not return a `bufid`. Instead, it returns the actual values of `tid`, `msgtag`, and `cnt` in `rtid, rtag, rcnt` respectively.

```
int (*old)() = pvm_recvf(int (*new)(int buf, int tid, int tag))
```

The routine pvm_recvf() modifies the receive context used by the receive functions and can be used to extend PVM. The default receive context is to match on source and message tag. This can be modified to any user defined comparison function. (See Appendix A for an example of creating a probe function with pvm_recf().) There is no Fortran interface routine for pvm_recvf().

5.6.4. Unpacking Data

The following C routines unpack (multiple) data types from the active receive buffer. In an application they should match their corresponding pack routines in type, number of items, and stride. `nitem` is the number of items of the given type to unpack, and `stride` is the stride.

```
int info = pvm_upkbyte(   char   *cp, int nitem, int stride )
int info = pvm_upkcplx(   float  *xp, int nitem, int stride )
int info = pvm_upkdcplx(  double *zp, int nitem, int stride )
int info = pvm_upkdouble( double *dp, int nitem, int stride )
int info = pvm_upkfloat(  float  *fp, int nitem, int stride )
int info = pvm_upkint(    int    *np, int nitem, int stride )
int info = pvm_upklong(   long   *np, int nitem, int stride )
int info = pvm_upkshort(  short  *np, int nitem, int stride )
int info = pvm_upkuint(  unsigned int   *np, int nitem, int stride )
int info = pvm_upkushort(unsigned short *np, int nitem, int stride )
int info = pvm_upkulong( unsigned long  *np, int nitem, int stride )
int info = pvm_upkstr(    char   *cp )
```

```
int info = pvm_unpackf( const char *fmt, ... )
```

The routine pvm_unpackf() uses a printf-like format expression to specify what and how to unpack data from the receive buffer.

A single Fortran subroutine handles all the unpacking functions of the above C routines.

```
call pvmfunpack( what, xp, nitem, stride, info )
```

The argument `xp` is the array to be unpacked into. The integer argument `what` specifies the type of data to be unpacked. (Same `what` options as for pvmfpack()).

6. Dynamic Process Groups

The dynamic process group functions are built on top of the core PVM routines. There is a separate library `libgpvm3.a` that must be linked with user programs that make use of any of the group functions. The pvmd does not perform the group functions. This is handled by a group server that is automatically started when the first group function is invoked. There is some debate about how groups should be handled in a message passing interface. There are efficiency and reliability issues. There are tradeoffs between static verses dynamic groups. And some people argue that only tasks in a group can call group functions.

In keeping with the PVM philosophy, the group functions are designed to be very general and transparent to the user at some cost in efficiency. Any PVM task can join or leave any group at any time without having to inform any other task in the affected groups. Tasks can broadcast messages to groups of which they are not a member. And in general any PVM task may call any of the following group functions at any time. The exceptions are pvm_lvgroup(), pvm_barrier(), and pvm_reduce() which by their nature require the calling task to be a member of the specified group.

```
int inum  = pvm_joingroup( char *group )
int info  = pvm_lvgroup( char *group )
```

```
call pvmfjoingroup( group, inum )
call pvmflvgroup( group, info )
```

These routines allow a task to join or leave a user named group. The first call to pvm_joingroup() creates a group with name `group` and puts the calling task in this group. pvm_joingroup() returns the instance number (`inum`) of the process in this group. Instance numbers run from 0 to the number of group members minus 1. In PVM 3 a task can join multiple groups.

If a process leaves a group and then rejoins it that process may receive a different instance number. Instance numbers are recycled so a task joining a group will get the lowest available instance number. But if multiple tasks are joining a group there is no guarantee that a task will be assigned its previous instance number.

To assist the user in maintaining a contiguous set of instance numbers despite joining and leaving, the pvm_lvgroup() function does not return until the task is confirmed to have left. A pvm_joingroup() called after this return will assign the vacant instance number to the new task. It is the users responsibility to maintain a contiguous set of instance numbers if his algorithm requires it. If several tasks leave a group and no tasks join, then there will be gaps in the instance numbers.

```
int tid = pvm_gettid( char *group, int inum )
```

```
call pvmfgettid( group, inum, tid )
```

The routine pvm_gettid() returns the `tid` of the process with a given group name and instance number. pvm_gettid() allows two tasks with no knowledge of each other to get each other's tid simply by joining a common group.

```
int inum = pvm_getinst( char *group, int tid )
```

```
call pvmfgetinst( group, tid, inum )
```

The routine pvm_getinst() returns the instance number of `tid` in the specified group.

```
int size = pvm_gsize( char *group )
```

```
call pvmfgsize( group, size )
```

The routine pvm_gsize() returns the number of members in the specified group.

```
int info = pvm_barrier( char *group, int count )
```

```
call pvmfbarrier( group, count, info )
```

On calling pvm_barrier() the process blocks until `count` members of a group have called pvm_barrier. In general `count` should be the total number of members of the group. A count is required because with dynamic process groups PVM can not know how many members are in a group at a given instant. It is an error for processes to call pvm_barrier with a group it is not a member of. It is also an error if the count arguments across a given barrier call do not match. For example it is an error if one member of a group calls pvm_barrier() with a count of 4, and another member calls pvm_barrier() with a count of 5.

```
int info = pvm_bcast( char *group, int msgtag )
```

```
call pvmfbcast( group, msgtag, info )
```

pvm_bcast() labels the message with an integer identifier `msgtag` and broadcasts the message to all tasks in the specified group except itself (if it is a member of the group).

For pvm_bcast() "all tasks" is defined to be those tasks the group server thinks are in the group when the routine is called. If tasks join the group during a broadcast they may not receive the message. If tasks leave the group during a broadcast a copy of the message will still be sent to them.

```
int info = pvm_reduce(  void (*func)(), void *data,
                        int nitem, int datatype,
                        int msgtag, char *group, int root )
```

```
call pvmfreduce(  func, data, count, datatype,
                  msgtag, group, root, info )
```

pvm_reduce() performs a global arithmetic operation across the group, for example, global sum or global max. The result of the reduction operation is returned on `root`. PVM supplies four predefined functions that the user can place in `func`. These are:

```
PvmMax
PvmMin
PvmSum
PvmProduct
```

The reduction operation is performed element-wise on the input data. For example, if the data array contains two floating point numbers and func is PvmMax, then the result contains two numbers – the global maximum of each group member's first number and the global maximum of each member's second number.

In addition users can define their own global operation function to place in `func`. See Appendix A for details. An example is given in `PVM_ROOT/examples/gexample`.

[Note: pvm_reduce() does not block. If a task calls pvm_reduce and then leaves the group before the root has called pvm_reduce an error may occur.]

7. Examples in C and Fortran

This section contains two example programs each illustrating a different way to organize applications in PVM 3. The examples have been purposely kept simple to make them easy to understand and explain. Each of the programs is presented in both C and Fortran for a total of four listings. These examples and a few others are supplied with the PVM source in `PVM_ROOT/examples`.

The first example is a master/slave model with communication between slaves. The second example is a single program multiple data (SPMD) model.

In a master/slave model the master program spawns and directs some number of slave programs which perform computations. PVM is not restricted to this model. For example, any PVM task can initiate processes on other machines. But a master/slave model is a useful programming paradigm and simple to illustrate. The master calls pvm_mytid(), which as the first PVM call, enrolls this task in the PVM system. It then calls pvm_spawn() to execute a given number of slave programs on other machines in PVM. The master program contains an example of broadcasting messages in PVM. The master broadcasts to the slaves the number of slaves started and a list of all the slave tids. Each slave program calls pvm_mytid() to determine their task ID in the virtual machine, then uses the data broadcast from the master to create a unique ordering from 0 to nproc minus 1.

Subsequently, pvm_send() and pvm_recv() are used to pass messages between processes.

When finished, all PVM programs call pvm_exit() to allow PVM to disconnect any sockets to the processes, flush I/O buffers, and to allow PVM to keep track of which processes are running.

In the SPMD model there is only a single program, and there is no master program directing the computation. Such programs are sometimes called *hostless* programs. There is still the issue of getting all the processes initially started. In example 2 the user starts the first copy of the program. By checking pvm_parent(), this copy can determine that it was not spawned by PVM and thus must be the first copy. It then spawns multiple copies of itself and passes them the array of tids. At this point each copy is equal and can work on its partition of the data in collaboration with the other processes. Using pvm_parent precludes starting the SPMD program from the PVM console because pvm_parent will return the tid of the console. This type of SPMD program must be started from a UNIX prompt.

```
#include "pvm3.h"
#define SLAVENAME "slave1"

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

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

    /* start up slave tasks */
    puts("How many slave programs (1-32)?");
    scanf("%d", &nproc);

    pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);

    /* Begin User Program */
    n = 100;
    initialize_data( data, n );

    /* Broadcast initial data to slave tasks */
    pvm_initsend(PvmDataRaw);
    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\n",result[who],who);
    }
    /* Program Finished exit PVM before stopping */
    pvm_exit();
}
```

Figure 3: C version of master example.

```
#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();
}
```

Figure 4: C version of slave example.

```
      program master1
c INCLUDE FORTRAN PVM HEADER FILE
      include 'fpvm3.h'

      integer i, info, nproc, numt, msgtype, who, mytid, tids(0:32)
      double precision result(32), data(100)
      character*12 nodename, arch

c     Enroll this program in PVM
      call pvmfmytid( mytid )

c     Initiate nproc instances of slave1 program
      print *,'How many slave programs (1-32)?'
      read *, nproc
      nodename = 'slave1'
      call pvmfspawn( nodename, PVMDEFAULT, '*', nproc, tids, numt )

c ------- Begin user program --------
      n = 100
      call initiate_data( data, n )

c     Broadcast data to all node programs
      call pvmfinitsend( 0, info )
      call pvmfpack( INTEGER4, nproc, 1, 1, info )
      call pvmfpack( INTEGER4, tids, nproc, 1, info )
      call pvmfpack( INTEGER4, n, 1, 1, info )
      call pvmfpack( REAL8,    data, n, 1, info )
      msgtype  = 1
      call pvmfmcast( nproc, tids, msgtype, info )

c     Wait for results from nodes
      msgtype  = 2
      do 30 i=1,nproc
         call pvmfrecv( -1, msgtype, info )
         call pvmfunpack( INTEGER4, who, 1, 1, info )
         call pvmfunpack( REAL8, result(who+1), 1, 1, info )
 30   continue
c ------------ End user program --------
c     Program finished leave PVM before exiting
      call pvmfexit()
      stop
      end
```

Figure 5: Fortran version of master example.

```
      program slave1
c INCLUDE FORTRAN PVM HEADER FILE
      include 'fpvm3.h'

      integer  info, mytid, mtid, msgtype, me, tids(0:32)
      double precision result, data(100)
      double precision work

c     Enroll this program in PVM
      call pvmfmytid( mytid )
c     Get the master's task id
      call pvmfparent( mtid )

c ------- Begin user program --------

c     Receive data from master
      msgtype  = 1
      call pvmfrecv( mtid, msgtype, info )
      call pvmfunpack( INTEGER4, nproc, 1, 1, info )
      call pvmfunpack( INTEGER4, tids, nproc, 1, info )
      call pvmfunpack( INTEGER4, n, 1, 1, info )
      call pvmfunpack( REAL8,    data, n, 1, info )

c     Determine which slave I am (0 -- nproc-1)
      do 5 i=0, nproc
         if( tids(i) .eq. mytid ) me = i
  5   continue

c     Do calculations with data
      result = work( me, n, data, tids, nproc )

c     Send result to master
      call pvmfinitsend( PVMDEFAULT, info )
      call pvmfpack( INTEGER4, me, 1, 1, info )
      call pvmfpack( REAL8,    result, 1, 1, info )
      msgtype  = 2
      call pvmfsend( mtid, msgtype, info )

c --------- End user program --------

c     Program finished. Leave PVM before exiting
      call pvmfexit()
      stop
      end
```

Figure 6: Fortran version of slave example.

```
#define NPROC 4
#include "pvm3.h"
main()
{
    int mytid, tids[NPROC], me, i;

    mytid = pvm_mytid();    /* ENROLL IN PVM */
    tids[0] = pvm_parent(); /* FIND OUT IF I AM PARENT OR CHILD */
    if( tids[0] < 0 ){      /* THEN I AM THE PARENT */
       tids[0] = mytid;
       me = 0;              /* START UP COPIES OF MYSELF */
       pvm_spawn("spmd", (char**)0, 0, "", NPROC-1, &tids[1]);
       pvm_initsend( PvmDataDefault );   /* SEND TIDS ARRAY */
       pvm_pkint(tids, NPROC, 1);        /* TO CHILDREN     */
       pvm_mcast(&tids[1], NPROC-1, 0);
    }
    else{                    /* I AM A CHILD */
       pvm_recv(tids[0], 0); /* RECEIVE TIDS ARRAY */
       pvm_upkint(tids, NPROC, 1);
       for( i=1; i<NPROC ; i++ )
          if( mytid == tids[i] ){ me = i; break; }
     }
/* All NPROC tasks are equal now
 * and can address each other by tids[0] thru tids[NPROC-1]
 * for each process 'me' is process index [0-(NPROC-1)]
 *-----------------------------------------------------------*/
     dowork( me, tids, NPROC );
     pvm_exit();             /* PROGRAM FINISHED EXIT PVM */
}
dowork( me, tids, nproc )    /* DOWORK PASSES A TOKEN AROUND A RING */
     int me, *tids, nproc;
{
     int token, dest, count=1, stride=1, msgtag=4;
     if( me == 0 ) {
        token = tids[0];
        pvm_initsend( PvmDataDefault );
        pvm_pkint( &token, count, stride );
        pvm_send( tids[me+1], msgtag );
        pvm_recv( tids[nproc-1], msgtag );
     }
     else {
        pvm_recv( tids[me-1], msgtag );
        pvm_upkint( &token, count, stride );
        pvm_initsend( PvmDataDefault );
        pvm_pkint( &token, count, stride );
        dest = (me == nproc-1)? tids[0] : tids[me+1] ;
        pvm_send( dest, msgtag );
     }
}
```

Figure 7: C version of SPMD example.

```
      program spmd
c INCLUDE FORTRAN PVM HEADER FILE
      include 'fpvm3.h'

      PARAMETER( NPROC=4 )
      integer mytid, me, numt, i
      integer tids(0:NPROC)
c                            ENROLL IN PVM
      call pvmfmytid( mytid )
c                            FIND OUT IF I AM PARENT OR CHILD
      call pvmfparent(tids(0))
      if( tids(0) .lt. 0 )  then
        tids(0) = mytid
        me = 0
c                            START UP COPIES OF MYSELF
        call pvmfspawn( 'spmd', PVMDEFAULT, '*', NPROC-1, tids(1), numt)
c                            SEND TIDS ARRAY TO CHILDREN
        call pvmfinitsend( 0, info )
        call pvmfpack( INTEGER4, tids, NPROC, 1, info )
        call pvmfmcast( NPROC-1, tids(1), 0, info )
      else
c                            RECEIVE THE TIDS ARRAY AND SET ME
        call pvmfrecv( tids(0), 0, info )
        call pvmfunpack( INTEGER4, tids, NPROC, 1, info )
        do 30 i=1, NPROC-1
          if( mytid .eq. tids(i) ) me = i
   30   continue
      endif
c-------------------------------------------------------------
c  all NPROC tasks are equal now
c  and can address each other by tids(0) thru tids(NPROC-1)
c  for each process me => process number [0-(NPROC-1)]
c-------------------------------------------------------------
      call dowork( me, tids, NPROC )

c                         PROGRAM FINISHED EXIT PVM
      call pvmfexit()
      stop
      end
```

Figure 8: Fortran version of SPMD example (part 1).

```
      subroutine dowork( me, tids, nproc )
      include 'fpvm3.h'
c--------------------------------------------------
c Simple subroutine to pass a token around a ring
c--------------------------------------------------
      integer me, nproc, tids( 0:nproc)
      integer token, dest, count, stride, msgtag
      count  = 1
      stride = 1
      msgtag = 4

      if( me .eq. 0 ) then
         token = tids(0)
         call pvmfinitsend( 0, info )
         call pvmfpack( INTEGER4, token, count, stride, info )
         call pvmfsend( tids(me+1), msgtag, info )
         call pvmfrecv( tids(nproc-1), msgtag, info )
      else
         call pvmfrecv( tids(me-1), msgtag, info )
         call pvmfunpack( INTEGER4, token, count, stride, info )
         call pvmfinitsend( 0, info )
         call pvmfpack( INTEGER4, token, count, stride, info )
         dest = tids(me+1)
         if( me .eq. nproc-1 ) dest = tids(0)
         call pvmfsend( dest, msgtag, info )
      endif
      return
      end
```

Figure 9: Fortran version of SPMD example (part 2).

8. Writing Applications

Application programs view PVM as a general and flexible parallel computing resource that supports a message-passing model of computation. This resource may be accessed at three different levels: the *transparent* mode in which tasks are automatically executed on the most appropriate hosts (generally the least loaded computer), the *architecture-dependent* mode in which the user may indicate specific architectures on which particular tasks are to execute, and the *low-level* mode in which a particular host may be specified. Such layering permits flexibility while retaining the ability to exploit particular strengths of individual machines on the network.

Application programs under PVM may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, the processes in existence may have arbitrary relationships between each other and in addition, any process may communicate and/or synchronize with any other. This allows for the most general form of MIMD parallel computation, but in practice most concurrent applications are more structured. Two typical structures are the SPMD model in which all processes are identical and the master/slave model in which a set of computational slave processes performs work for one or more master processes.

8.1. General performance considerations

There are no limitations to the programming paradigm a PVM user may choose. Any specific control and dependency structure may be implemented under the PVM system by appropriate use of PVM constructs. On the other hand there are certain considerations that the application developer should be aware when programming any message passing system.

The first consideration is task granularity. This is typically measured as a ratio of the number of bytes received by a process to the number of floating point operations a process performs. By doing some simple calculations of the computational speed of the machines in a PVM configuration and the available network bandwidth between the machines, a user can get a rough lower bound on the task granularity to be used in an application. The tradeoff is the larger the granularity the higher the speedup but often a reduction in the available parallelism as well.

The second consideration is the number of messages sent. The number of bytes received may be sent in many small messages or in a few large messages. While using a few large messages reduces the total message start-up time, it may not cause the overall execution time to decrease. There are cases where small messages can be overlapped with other computation so that their overhead is masked. The ability to overlap communication with computation and the optimal number of messages to send are application dependent.

A third consideration is whether the application is better suited to functional parallelism or data parallelism. We define functional parallelism to be different machines in a PVM configuration performing different tasks. For example, a vector supercomputer may solve a part of a problem suited for vectorization, a multiprocessor may solve another part of the problem that is suited to parallelization, and a graphics workstation may be used to visualize the generated data in real time. Each machine performs different functions (possibly on the same data).

In the data parallelism model, the data is partitioned and distributed to all the machines in the PVM configuration. Operations (often similar) are performed on each set of data and information is passed between processes until the problem is solved. Data parallelism has been popular on distributed-memory multiprocessors because it requires writing only one parallel program that is executed on all the machines, and because it can often be scaled up to hundreds of processors. Many linear algebra, PDE, and matrix algorithms have been developed using the data parallelism model.

Of course in PVM both models can be mixed in a hybrid that exploits the strengths of each machine. For example the parallel code that runs on the multiprocessor in the above functional example may itself be written in PVM using a data parallelism model.

8.2. Network particular considerations

There are additional considerations for the application developer if he wishes to run his parallel application over a network of machines. His parallel program will be sharing the network with other users. This multiuser, multitasking environment affects both the communication and computational performance of his program in complex ways.

First consider the effects of having different computational power on each machine in the configuration. This can be due to having a heterogeneous collection of machines in the virtual machine which differ in their computational rates. Just between different brands of workstations there can be two orders of magnitude difference in power. For supercomputers there can be even more. But even if the user specifies a homogeneous collection of machines, he can see large differences in the available performance on each machine. This is caused by the multitasking of his own or other user's tasks on a subset of the configured machines. If the user divides his problem into identical pieces one for each machine, (a common approach to parallelization), then the above consideration may adversely effect his performance. His application will run as slow as the task on the slowest machine. If the tasks coordinate with each other, then even the fast machines will be slowed down waiting for the data from the slowest tasks.

Second consider the effects of long message latency across the network. This could be caused by the distance between machines if a wide-area network is being employed. It can also be caused by contention on your local network from your own program or other users. Consider that Ethernet networks are a bus. As such only one message can be on the bus at any time. If the application is designed so that each of its tasks only sends to a neighboring task then one might assume there would be no contention. On a distributed memory multiprocessor, such as an Intel Paragon, there would be no contention and all the sends could proceed in parallel. But over Ethernet the sends will be serialized leading to varying delays (latencies) in the messages arriving at neighboring tasks. Other networks such as token ring, FIDDI, and HiPPI, all have properties that can cause varying latencies. The user should determine if latency tolerance should be designed into his algorithm.

Third consider that the computational performance and effective network bandwidth are dynamicly changing as other users share these resources. An application may get a very good speedup during one run and a poor speedup on a run just a few minutes later. During a run an application can have its normal synchronization pattern thrown off causing some tasks to wait for data. In the worst case, a synchronization error could exist in an application that only shows up when the dynamic machine loads fluctuate in a particular way. Because such conditions are difficult to reproduce, these types of errors can be very hard to find.

Many of these network considerations are taken care of by incorporating some form of load balancing into a parallel application. The next section describes some of the popular load balancing methods.

8.3. Load Balancing

In a multiuser network environment we have found that load balancing can be the single most important performance enhancer. [6]. There are many load balancing schemes for parallel programs. In this section we will describe the three most common schemes used in network computing.

The simplest method is *static* load balancing. In this method the problem is divided up, and tasks are assigned to processors only once. The data partitioning may occur off-line before the job is started, or the partitioning may occur as an early step in an application. The size of the tasks or the number of tasks assigned to a given machine can be varied to account for the different computational powers of the machines. Since all the tasks can be active from the beginning, they can communicate and coordinate with one another. On a lightly loaded network, static load balancing can be quite effective.

When the computational loads are varying a *dynamic* load balance scheme is required. The most popular method is called the *Pool of Tasks* paradigm. It is typically implemented in a master/slave program where the master program creates and holds the "pool" and farms out tasks to slave programs as they fall idle. The pool is usually implemented as a queue and if the tasks vary in size then the larger tasks are placed near the head of the queue. With this method all the slave processes are kept busy as long as there are tasks left in the pool. An example of the Pool of Tasks paradigm can be seen in the `xep` program supplied with the PVM source under `pvm3/xep`. Since tasks start and stop at arbitrary times with this method, it is better suited to applications which require no communication between slave programs and only communication to the master and files.

A third load balance scheme which doesn't use a master process requires that at some predetermined time all the processes will reexamine and redistribute their work loads. An example is in the solution of nonlinear PDEs, each linearized step could be statically load balanced and between each linear step the processes examine how the problem has changed and redistribute the mesh points. There are several variations of this basic scheme. Some implementations never synchronize with all the processes but instead distribute excess load only with their neighbors. Some implementations wait until a process signals that its load balance has gotten above some tolerance before going through a load redistribution rather than waiting on a fixed time.

9. Debugging Methods

In general, debugging parallel programs is much more difficult than debugging serial programs. Not only are there more processes running simultaneously, but their interaction can also cause errors. For example a process may receive the wrong data that later causes it to divide by zero. Another example is *deadlock* where a programming error has caused all the processes to be waiting on messages. All PVM routines return an error condition if some error has been detected during their execution. A list of these codes and their meaning is given in Table 2.

ERROR CODE		MEANING
`PvmOk`	0	okay
`PvmBadParam`	-2	bad parameter
`PvmMismatch`	-3	barrier count mismatch
`PvmNoData`	-5	read past end of buffer
`PvmNoHost`	-6	no such host
`PvmNoFile`	-7	no such executable
`PvmNoMem`	-10	can't get memory
`PvmBadMsg`	-12	can't decode received msg
`PvmSysErr`	-14	pvmd not responding
`PvmNoBuf`	-15	no current buffer
`PvmNoSuchBuf`	-16	bad message id
`PvmNullGroup`	-17	null group name is illegal
`PvmDupGroup`	-18	already in group
`PvmNoGroup`	-19	no group with that name
`PvmNotInGroup`	-20	not in group
`PvmNoInst`	-21	no such instance in group
`PvmHostFail`	-22	host failed
`PvmNoParent`	-23	no parent task
`PvmNotImpl`	-24	function not implemented
`PvmDSysErr`	-25	pvmd system error
`PvmBadVersion`	-26	pvmd-pvmd protocol mismatch
`PvmOutOfRes`	-27	out of resources
`PvmDupHost`	-28	host already configured
`PvmCantStart`	-29	failed to exec new slave pvmd
`PvmAlready`	-30	slave pvmd already running
`PvmNoTask`	-31	task does not exist
`PvmNoEntry`	-32	no such (group,instance)
`PvmDupEntry`	-33	(group,instance) already exists

Table 2: Error codes returned by PVM 3 routines.

By default PVM prints error conditions detected in PVM routines. The routine pvm_setopt() allows the user to turn this automatic reporting off. Diagnostic prints from spawned tasks can be viewed using the PVM console redirection or by calling pvm_catchout() in the spawning task (often the master task). pvm_catchout() causes the standard output of all subsequently spawned tasks to appear on the standard output of the spawner.

PVM tasks can be started manually under any standard serial debugger, for example dbx. stdout from tasks started manually always appears in the window in which it was started.

PVM tasks that are spawned can also be started under a debugger. By setting the `flag` option to include **PvmTaskDebug** in the pvm_spawn() call, by default PVM will execute the shell script `PVM_ROOT/lib/debugger`. As supplied this script starts an xterm window on the host PVM was started on and spawns the task under a debugger in this window. The task being debugged can be executed on any of the hosts in the virtual machine as specified by the `flag` and `where` arguments in pvm_spawn(). The user can create his own personalized debugger script to include a preferred debugger or even a parallel debugger if one is available. The user can then tell PVM where to find this script by using the `bx=` option in the hostfile.

Diagnostic print statements sent to stderr from a spawned task will not appear on the user's screen. All these prints are routed to a single log file of the form `/tmp/pvml.<uid>` on the host where PVM was started. stdout statements may appear in this file as well although I/O buffering may make this a less useful debugging method. Tasks that are spawned from the PVM console can have their stdout (and all their children's stdout) redirected back to the console window or to a separate file.

The routine pvm_setopt() also allows the user to set a debug mask which determines the level of debug messages to be printed to `/tmp/pvml.<uid>`. By default the debug level is set to 'no debug messages'. The debug level can be changed multiple times inside an application to debug a single routine or section of code. The debug statements describe only what PVM is doing and not what the application is doing. The user must infer what the application was doing from the PVM debug statements. This may or may not be reasonable depending on the nature of the bug.

Experience has led to the following three steps in trying to debug PVM programs. First, if possible, run the program as a single process and debug as any other serial program. The purpose of this step is to catch indexing and logic errors unrelated to parallelism. Once these errors are corrected, go to the second step.

Second, run the program using 2–4 processes on a single machine. PVM will multitask these processes on the single machine. The purpose of this step is to check the communication syntax and logic. For example was a message tag of 5 used in the send but the receiver is waiting for a message with tag equal to 4. A more common error to discover at this step is the use of non-unique message tags. To illustrate assume that the same message tag is always used. A process receives some initial data in three separate messages, but it has no way of determining which of the messages contains what data. PVM returns any message that matches the requested source and tag, so it is up to the user to make sure that this pair uniquely identifies the contents of a message. The non-unique tags error is often very hard to debug because it is sensitive to subtle synchronization effects and may not be reproducible from run to run. If the error can not be determined by the PVM error codes or from a quick print statement, then the user can get complete debugger control of his program by starting one or all of his tasks under debuggers. This allows break points, variable tracing, single stepping, and trace backs for each process even while it passes messages back and forth to other PVM tasks that may or may not be running under dbx.

The third step is to run the same 2–4 processes across several machines. The purpose of this step is to check for synchronization errors that are produced by network delays. The kind of errors often discovered at this step are sensitivity of the algorithm to message arrival order, and program deadlock caused by logic errors sensitive to network delays. Again complete debugger control can be obtained in this step, but it may not be as useful because the debugger may shift or mask the timing errors observed earlier.

10. Implementation Details

This section gives a glimpse at the design goals and implementation details of the single-cpu UNIX (generic) version of PVM. A complete technical description of PVM can be found in [4].

There were three main goals under consideration while building version 3. We wanted the virtual machine to be able to scale to hundreds of hosts and thousands of tasks. This requires efficient message-passing operations and, more importantly, operations (such as task management) to be as localized as possible in order to avoid bottlenecks.

We wanted the system to be portable to any version of UNIX and also to machines not running UNIX, especially MPPs (message passing machines with many processor nodes).

Finally, we wanted the system to be able to withstand host and network failures, allowing fault-tolerant applications to be built.

In order to keep PVM as portable as possible, we avoided the use of operating system or programming language features that would be be hard to retrofit if unavailable. We decided not to use multi-threaded code, or more specifically, not to overlap I/O and processing in tasks. Many UNIX machines have light- or heavy-weight thread packages or asynchronous I/O system calls, but these are variable enough that many code changes would be required. On machines where threads are not available, it's possible to use signal-driven I/O and interrupt handlers to move data semi-transparently while computing. This solution would be even more difficult to maintain, partly due to differences between various systems, but mainly because the signal mechanism is not appropriate for the task.

While the generic port is kept as simple as possible, PVM can still be optimized for any particular machine. As facilities like threads become more standardized, we expect to make use of them.

We assume that sockets are available for interprocess communication and that each host in a virtual machine group can connect directly to every other host using IP protocols (TCP and UDP). That is, the pvmd expects to be able to send a packet to another pvmd in a single hop. The requirement of full IP connectivity could presumably be removed by specifying routes and allowing the pvmds to forward messages. Note that some MPP machines don't make sockets available on the processing nodes, but do have them on the front-end (where the pvmd runs).

10.1. Task Identifiers

PVM uses a 32-bit integer called a task identifier (TID) to address pvmds, tasks, and groups of tasks within a virtual machine. A TID identifies a unique object within its entire virtual machine, however, TIDs are recycled when no longer in use.

The TID contains four fields as shown in Figure 10. It is currently partitioned as indicated, however the sizes of the fields could someday change (possibly dynamically as a virtual machine is configured). Since the TID is used so heavily, it is designed to fit into the largest integer data type available on a wide range of machines.

```
   3                   2                   1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+---------------+---------------+---------------+---------------+
|               |               |               |               |
+---------------+---------------+---------------+---------------+
 S G \_________ H _________/ \_______________ L _______________/
```

Figure 10: Generic Task ID

The fields S, G and H have meaning globally, that is, each pvmd of a virtual machine interprets them the same way. The H field contains a host number relative to the virtual machine. As it starts up, each pvmd is configured with a unique nonzero host number and therefore "owns" part of the address space of the machine. Host number zero is used, depending on context, to refer either to the local pvmd or to a "shadow" pvmd (called pvmd'), of the master pvmd. The maximum number of hosts in a virtual machine is limited to $2^H - 1$ (4095). The mapping between host numbers and hosts is known to each pvmd.

The S field is a historical leftover, and causes slightly schizoid naming. Messages are addressed to a pvmd by setting the S bit and the host field, and zeroing the L field. In the future, this bit should be reclaimed to make the H or L space larger.

Each pvmd is allowed to assign local meaning to the L field (when the H field is set to its own host number), with the exception that all bits cleared is reserved to mean the pvmd itself. In the generic UNIX port, L field values are assigned by a counter, and the pvmd maintains a map between L values and UNIX process IDs. As with the number of hosts, the number of tasks per host is limited by the size of its TID field. Since the L field is allotted 18 bits, at most 262143 tasks can exist concurrently on a host.

In multiprocessor ports the L field is often subdivided, for example into a partition field (P), a node number (N) field and a location bit (W) (Figure 11).

The P field specifies a machine partition (sometimes called a "process type" or "job"), in the case where the pvmd can manage multiple MPP partitions. The N field determines a specific cpu node in the partition. The W bit indicates whether a task is

```
  3                   2                   1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+---------------+---------------+---------------+---------------+
|               |               |               |               |
+---------------+---------------+---------------+---------------+
 S G \_________ H _________/ W \___ P ___/ \________ N ________/
```

Figure 11: MPP Task ID

running on an MPP (compute) node or the host (service node) processor. The setting of the W bit can be determined by the "ps -a" output from the PVM console. Since the TID output by ps is a hexidecimal number, the fifth digit from the right contains the W bit. The following is a simple state table to determine if the W bit is set to 0 or 1:

W bit	task running on	contents of 5th tid digit
0	MPP compute node	0,1,4,5,8,9,c,d
1	host/service node	2,3,6,7,a,b,e,f

For example, if your TID is 60001, then you know that your task is running on an MPP compute node."

The design of the TID enables the implementation to meet some of the goals stated earlier. Tasks can be assigned TIDs by their local pvmds without off-host communication, eliminating a bottleneck at an ID server. Messages can be routed to a destination from anywhere in the system, thanks to the hierarchical naming. Portability is enhanced because the L field can be redefined easily. Finally, space is reserved for error codes. When a function can return a vector of TIDs mixed with error codes, it is useful if the error codes don't correspond to legal TIDs.

The TID space is divided up as follows:

	_S_	_G_	_H_	_L_
Task identifier	0	0	1..maxhost	1..maxlocal
Pvmd identifier	1	0	1..maxhost	0
Local pvmd (from task)	1	0	0	0
Pvmd' from master pvmd	1	0	0	0
Multicast address	0	1	1..maxhost	x
Error code	1	1	< small negative number >	

Naturally, TIDs are intended to be opaque to the application and the programmer should not attempt to predict their values or modify them without using functions supplied with the programming library. More structured naming (from the application programming standpoint) can be obtained by using a name server library layered on top of the raw PVM calls, if the convenience is deemed worth the cost of name lookup.

10.2. The PVM Daemon

One pvmd runs on each host of a virtual machine, and the pvmds are configured to work together. Pvmds owned by (running as) one user do not interact with ones owned by others. The pvmd was designed to run under a nonprivileged user ID and serve a single user in order to reduce security risk, and to minimize the impact of one PVM user on another.

The pvmd doesn't do any computation, rather it serves as a message router and controller. It provides a point of contact on each host, both from inside and outside, as well as authentication, process control and fault detection. Idle pvmds occasionally ping each other to verify reachability, and ones that don't answer are marked dead. Pvmds are hopefully more survivable than application components, and will continue to run in the event of a program crash, to aid in debugging.

The first pvmd (started by hand) is designated the "master" pvmd, while the others (started by the master) are called "slaves". During most normal operations, all pvmds are considered equal. Only the master can start new slave pvmds and add them to the virtual machine configuration. Requests to reconfigure the machine originating on a slave host are forwarded to the master. Likewise, only the master can forcibly delete hosts from the machine. If the master pvmd loses contact with a slave, it marks the slave dead and deletes it from the configuration. If a slave pvmd loses contact with the master, the slave shuts itself down. This algorithm ensures that the virtual machine can't become partitioned and continue to run as two partial machines, like a worm cut in half. Unfortunately, this impacts fault tolerance because the master must never crash. There is currently no way for the master to hand off its duty to another pvmd, so it always remains part of the configuration.

The data structures of primary importance in the pvmd are the host and task tables, which describe the virtual machine configuration and track tasks running under the pvmd. Attached to these are queues of packets and messages, and "wait contexts" to hold state information for multitasking in the pvmd.

At startup time, a pvmd configures itself as either a master or slave, depending on its command line arguments. This is when it creates and binds sockets to talk to tasks and other pvmds, opens an error log file, and initializes tables. For a master pvmd, configuration may include reading the hostfile and determining default parameters, such as the host name. A slave pvmd gets its parameters from the command line and sends a line of data back to the starter process, for inclusion in the host table. If the master pvmd is given a file of hosts to be started automatically, it sends a `DM_ADDHOST` message to itself. Thus the slave hosts are brought into the configuration just as though they had been added dynamically. Slave pvmd startup is described in the next section.

After configuring itself, the pvmd enters a loop in function `work()`. At the core of the work loop is a call to `select()` that probes all sources of input for the pvmd (local tasks and the network). Incoming packets are received and routed to their destinations. Messages addressed to the pvmd are reassembled and passed to one of the entry points `loclentry()`, `netentry()` or `schedentry()`.

10.2.1. Pvmd Startup

Getting a slave pvmd started is a messy task with no good solution. The goal is to get a pvmd process running on the new host, with enough information (i.e. the identity of the master pvmd) to let it be fully configured and added as a peer.

Several different mechanisms are available, depending on the operating system and local installation. Naturally, we want to use a method that is widely available, secure, fast and easy to install. We'd like to avoid having to type passwords all the time, but don't want to put them in a file from where they can be stolen. No system meets all of these criteria. `Inetd` would give fast, reliable startup, but would require that a sysadmin install PVM on each host to be used. Connecting to an already-running pvmd or pvmd server at a reserved port number presents similar problems. Starting the pvmd with an `rlogin` or `telnet` "chat" script would allow access even to hosts with `rsh` services disabled or IP-connected hosts behind firewall machines, and would require no special privilege to install. The main drawback is the effort required to get the chat program and script working reliably. Two widely available systems are `rsh` and `rexec()`. We use both to cover most of the features required. In addition, a manual startup option allows the user to take the place of a chat program, starting the pvmd manually and typing in the configuration.

`rsh` is a privileged program which can be used by the pvmd to run commands on a foreign host without a password, provided the destination host can be made to trust the source host. This can be done either by making it equivalent (requires a sysadmin) or by creating a `.rhosts` file on the destination host. As `rsh` can be a security risk, it's use is often discouraged by disabling it or automatically removing `.rhosts` files. The alternative, `rexec()`, is a function compiled into the pvmd. Unlike `rsh`, which can't take a password, `rexec()` requires the user to supply one at run time, either by typing it in or placing it in a `.netrc` file (this is a really bad idea).

When the master pvmd gets a `DM_ADD` message, it creates a new host table entry for each requested host. It looks up the IP addresses and sets the options to default settings or copies them from advisory host table entries. The host descriptors are kept in a `waitc_add` structure attached to a wait context, and not yet added to the host table. Then, it forks a shadow pvmd (pvmd') to do the dirty work, passing it a list of hosts and commands to execute.

Any of several steps in the startup process (for example getting the host IP address, starting a shell) can block for seconds or minutes, and the master pvmd must be able to respond to other messages during this time. The shadow has host number 0 and communicates with the master through the normal pvmd-pvmd interface, though it never talks to the slave pvmds. Likewise, the normal host failure mechanism is used to provide fault recovery. The startup operation has a wait context in the master pvmd. In the event the shadow breaks, the master catches a SIGCHLD from it and calls `hostfailentry()`, which cleans up.

Pvmd' uses `rsh` or `rexec()` (or manual startup) to start a pvmd on each new host, pass it parameters and get a line of configuration information back from it. When finished, pvmd' sends a `DM_STARTACK` message back to the master pvmd, containing the configuration lines or error messages. The master parses the results and completes the host descriptors held in the wait context. Results are sent back to the originator in a DM_ADDACK message. New hosts successfully started are configured into the machine using the host table update (DM_HTUPD) protocol. The configuration dialog between pvmd' and a new slave is similar to the following:

```
pvmd' -> slave:
    (exec) $PVM_ROOT/lib/pvmd -s -d8 -nhonk 1 80a9ca95:0f5a 4096 3 80a95c43:0000

slave -> pvmd':
    ddpro<2312> arch<ALPHA> ip<80a95c43:0b3f> mtu<4096>

pvmd' -> slave:
    EOF
```

The parameters of the master pvmd (debug mask, host table index, IP address and MTU) and slave (host name, host table index and IP address) are passed on the command line. The slave replies with its configuration (pvmd-pvmd protocol revision number, host architecture, IP address and MTU). It waits for an EOF from pvmd' and disconnects from the pipe, putting itself in probationary running status (`runstate = PVMDSTARTUP`). If it receives the rest of its configuration information from the master pvmd within a timeout period (DDBAILTIME, by default five minutes) it comes up to normal running status. Otherwise, it assumes there is some problem with the master and exits.

If a special task, called a "hoster", has registered with the master pvmd prior to receipt of the DM_ADD request, the normal startup system is not used. Instead of forking the pvmd', a SM_STHOST message is sent to the "hoster" task. It must start the remote processes as described above (using any mechanism it wants), pass parameters and collect replies, then send a SM_STHOSTACK message back to the pvmd. So, the method of starting slave pvmds is dynamically replaceable, with a hoster that does not have to understand the configuration protocol. If the hoster task fails during an add operation, the pvmd uses the wait context to recover. It assumes that none of the processes were started and sends a DM_ADDACK message indicating a system error.

Note: Recent experience suggests that it would be cleaner to manage the shadow pvmd through the task interface instead of the host interface. This would more naturally allow multiple starters to run at once (the parallel startup is currently implemented explicitly in a single pvmd' process).

10.2.2. Host Table

A host table describes the configuration of a virtual machine. Host tables are usually synchronized across all pvmds in a virtual machine, although they may not be in agreement at all times. In particular, hosts are deleted by a pvmd from its own host table whenever it determines them to be unreachable (by timing out while trying to communicate). In other words, the machine configuration may decay over time as hosts crash or their networks become disconnected. If a pvmd knows it is being killed or panics, it may be able to notify its peers, so they know it is down without having to wait for a timeout.

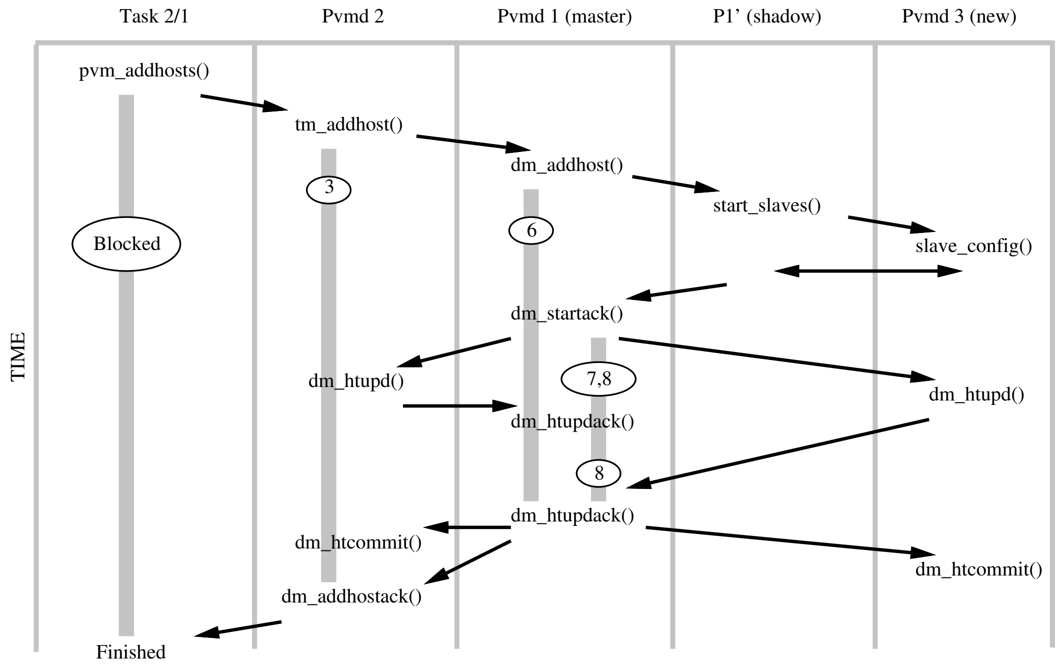

Figure 12: Addhost Timeline

The host tables of slave pvmds are modified on command from the master pvmd using `DM_HTUPD`, `DM_HTCOMMIT` and `DM_HTDEL` messages. The delete operation is very simple – on receiving a `DM_HTDEL` message, a pvmd calls the `hostfailentry()` function for each host listed in the message, as though the deleted pvmds have crashed. The add operation is done more carefully, with a three-phase commit, in order to guarantee global availability of the new hosts synchronously with completion of the add-host request. A task calls `pvm_addhost()`, which sends a request to the task's pvmd, which in turn sends a `DM_ADD` message to the master pvmd (possibly itself). The master pvmd uses its shadow process to start and configure the new slaves, then broadcasts a `DM_HTUPD` message. Upon receiving this message, each slave knows the identity of the new pvmd, and the new pvmd knows the identities of the previously existing ones. The master waits for an acknowledging `DM_HTUPDACK` message from every slave, then sends a `DM_ADDACK` reply to the original request, giving the new host ID. Finally, an `HT_COMMIT` message is broadcast, which commands the slaves to flush the old host table. When several hosts are added at once, the work is done in parallel, and the host table updated all at once, allowing the whole operation to take only slightly more time than for a single host.

Host descriptors (hostd) can be shared by multiple host tables, that is, each hostd has a refcount of how many host tables include it. As the configuration of the machine changes, the descriptor for each host (except ones added and deleted of course) remains the same.

Host tables serve multiple uses: They describe the configuration of the machine and hold packet queues and message buffers. They allow the pvmd to manipulate sets of hosts, for example when picking candidate hosts on which to spawn a task, or updating the virtual machine configuration. Also, the advisory host file supplied to the master pvmd is parsed and stored in a host table.

10.2.3. Task Table

Each pvmd maintains a list of all tasks under its management. Every task, regardless of state, is a member of a threaded list, sorted by `t_tid` (task ID). Most tasks are also kept in a second list, sorted by `t_pid`. In the generic port, `t_pid` holds the process ID of the task. The head of both lists is a dummy task descriptor, pointed to by global `locltasks`. Since the pvmd often needs to search for a task by TID or PID, it could be more efficient to maintain these two lists as self-balancing trees.

10.2.4. Wait Contexts

Wait contexts (waitcs) are used by the pvmd to hold state information when a thread of operation must be interrupted. The pvmd is not truly multi-threaded, but can perform operations concurrently. For example, when a pvmd gets a syscall from a task, it sometimes has to interact with another pvmd. Since it serves as a message router, it can't block while waiting for the foreign pvmd to respond. Instead, it saves any information specific to the syscall in a waitc and returns immediately to the `work()` loop. When the reply arrives, the pvmd uses the information stashed in the waitc to complete the syscall and reply to the task. Waitcs are numbered serialy, and the number is sent in the message header along with the request and returned with the reply.

For certain operations, the TIDs involved and the parameter `kind` are the only information saved. The waitc includes a few extra fields to handle most of the remaining cases, and a pointer, `wa_spec`, which can point to a block of extra data for special cases. These are the spawn and host startup operations, in which `wa_spec` points to a `struct waitc_spawn` or `struct waitc_add`.

Some operations require more than one phase of waiting – this can be in series or parallel, or even nested (if the foreign pvmd has to make another request). In the parallel case, a separate waitc is created for each foreign host. The individual waitcs are "peered" together to indicate they pertain to the same operation. Their `wa_peer` and `wa_rpeer` fields are linked together to form a list (with no sentinel node). If a waitc has no peers, its peer links point to itself, putting it in a group of one. Usually, all waitcs in a peer group share pointers to any common data, for example a `wa_spec` block. All existing multi-host parallel operations are conjunctions; a peer group of waitcs is finished waiting when every waitc in the group is finished. As replies come back, finished waitcs are collapsed out of the list and deleted. Finally, when the finished waitc is the only one in its group, the operation is complete.

When a foreign host fails or a task exits, the pvmd searches `waitlist` for any waitcs blocked on its TID. These are terminated, with differing results depending on the kind of wait. Waitcs blocking for the dead host or task are not deleted immediately. Instead, their `wa_tid` fields are zeroed to keep the wait ID active.

10.2.5. Fault Detection and Recovery

From the pvmd's point of view, fault tolerance means that it can detect when a foreign pvmd is down and recover without crashing. If the foreign pvmd was the master, however, it has to shut down. Otherwise, the pvmd itself doesn't care about host failures, except that it must complete any operations waiting on the dead hosts. From the task's point of view, fault detection means that any operation involving a down host will return an error condition, instead of simply hanging forever. It is left to the application programmer to use this capability wisely.

Fault detection originates in the pvmd-pvmd protocol, when a packet goes unacknowledged for three minutes. Function `hostfailentry()` is called, which scans `waitlist` and terminates any waits involving the failed host. (See Pvmd-Pvmd Communication section for details)

10.3. The Programming Library

The libpvm library is a collection of functions that allow a task to interface with the pvmd and other tasks. It contains functions for packing (composing) and unpacking messages, as well as ones that perform PVM "syscalls", using the message functions to send service requests to the pvmd and receive replies. It is intentionally kept as simple and small as possible. Since it shares address space with unknown, possibly buggy, code, it can be easily broken or subverted. Minimal sanity-checking of syscall parameters is performed, leaving further authentication to the pvmd.

The programming library is written in C and so naturally supports C and C++ applications. The Fortran library, `libfpvm3.a`, is also written in C and is a set of "wrapper" functions that conform to the Fortran calling conventions and call the C library functions. The Fortran/C linking requirements are portably met by preprocessing the C source code for the Fortran library with `m4` before compilation.

The top level of the libpvm library, including most of the programming interface functions, is written in a machine/operating system-independent style. The bottom level is kept separate and can be modified or replaced with a new machine-specific file when porting PVM to a new OS or MPP.

On the first call to (most) any libpvm function, that function calls `pvmbeatask()` to initialize the library state and connect the task to its pvmd. The details of connecting are slightly different between anonymous tasks (not spawned by the pvmd) and spawned tasks.

So that anonymous tasks can find it, the pvmd publishes the address of the socket where it listens for connections in `/tmp/pvmd.<uid>`, where *uid* is the numeric user ID under which the pvmd runs. This file contains a line such as "`7f000001:06f7`". As a shortcut, spawned tasks inherit environment variable `PVMSOCK`, containing the same information.

A spawned task needs a second bit of data to reconnect successfully, namely its expected process ID. When a task is spawned by the pvmd, a task descriptor (described earlier) is created for during the *exec* phase. The descriptor is necessary, for example, to stash any messages that arrive for the task before it's fully reconnected and ready to receive them. During reconnection, the task identifies itself to the pvmd by its PID. If the task is always the child of the pvmd, (i.e. the process *exec*'d by it) then it could use its PID as returned by `getpid()` to identify itself. To allow for intervening processes, such as debuggers, the pvmd passes an environment variable, `PVMEPID`, to the task, which uses that value in preference to its real PID. The task also passes its real PID so it can be controlled by the pvmd via signals.

So, `pvmbeatask()` creates a TCP socket and does a proper connection dance with the pvmd. They must each prove their identity to the other, to prevent a different user from spoofing the system. The pvmd and task each create a file in `/tmp` owned and writable only by their UID. They attempt to write in each others' files then check their own files for change. If successful, have proved their identities. Note this authentication is only as strong as the filesystem and the authority of root on each machine.

A protocol serial number (`TDPROTOCOL`, in `tdpro.h`) is compared whenever a task connects to its pvmd or another task. This number should be incremented whenever a change in the protocol makes it incompatible with the previous version.

Disconnecting is much simpler. It can be done forcibly by a *close* from either end, for example by exiting the task process. The function `pvm_exit()` performs a clean shutdown, such that the process can be connected again later (it would get a different TID).

10.4. Communication

We chose to base PVM communication on TCP and UDP Internet protocols. While other, more appropriate, protocols exist, they aren't as generally available, which would limit portability of the system. Another concession is that the PVM protocol drivers run as normal processes (pvmd and tasks), without modifications to the operating system. Naturally, the message-passing performance is degraded somewhat by this strategy. It's expensive to read timers and manage memory from user space, while extra context switches and copy operations are incurred. Performance would be better if the code was integrated into the kernel, or alternatively, if the network interface was made directly available to processes, bypassing the kernel. However, when running on Ethernet, the effects of this overhead seem to be minimal. Performance is determined more by the quality of the network code in the kernel. When running on faster networks, direct task-task routing improves performance by minimizing the number of hops.

This section explains where and how TCP and UDP are employed and describes the PVM protocols built on them. There are three connections to consider: Between pvmds, between a pvmd and its tasks, and between tasks.

10.4.1. Pvmd-Pvmd Communication

PVM daemons communicate with one another through UDP sockets. As UDP is an unreliable delivery service which can lose, duplicate or reorder packets, we need an acknowledgement and retry mechanism. UDP also imposes a limit on the length of a packet, which requires PVM to fragment long messages. Using UDP we built a reliable sequenced packet delivery service, and on top of that a message layer, providing a connection similar to a TCP stream, but with record bounds.

We considered using TCP, but three factors make it inappropriate. First, the virtual machine must be able to scale to hundreds of hosts. Each open TCP connection consumes a file descriptor in the pvmd, and some operating systems limit the number of open files to as few as 32. A single UDP socket can send to and receive from any number of remote UDP sockets. Next, a virtual machine composed of N hosts would need up to $N(N-1)/2$ connections, which would be expensive to establish. Since the identity of every host in the virtual machine is known, our protocol can be initialized to the correct state without a connect phase. Finally, the pvmd-pvmd packet service must be able to detect when foreign pvmds or hosts have crashed or the network has gone down. To accomplish this, we need to set timeouts in the protocol layer. While we might have used the TCP keepalive option, we don't have adequate control over the idle time between keepalives and timeout parameters.

All the parameters and default values for pvmd-pvmd communication are defined in file `ddpro.h`. Also defined there are the message codes for the various pvmd entry points (`DM_XXX`). A serial number (`DDPROTOCOL`) is checked whenever a pvmd is added to the virtual machine. It must be incremented whenever a change is made to the protocol that makes it incompatible with previous versions.

The headers for packets and messages are shown in Figures 13 and 14. Multi-byte values are sent in "network byte order", that is, most significant byte first.

```
 Byte 0          1               2               3
+---------------+---------------+---------------+---------------+
|                        Destination TID                        |
+---------------+---------------+---------------+---------------+
|                           Source TID                          |
+---------------+---------------+---------------+---------------+
|        Sequence Number        |          Ack Number           |
+---------------+---------------+---------------+---------------+
| | | |A|F|D|E|S|                                               |
| | | |C|I|A|O|O|                    Unused                     |
| | | |K|N|T|M|M|                                               |
+---------------+---------------+---------------+---------------+
/                                                               /
```

Figure 13: Pvmd-pvmd Packet Header

The source and destination fields hold the TIDs of the true source and final destination of packet, regardless of the route it takes.

Sequence and acknowledgement numbers start at 1 and increment to 65535, then wrap around to zero. They are initialized in the host table for new hosts so that the connection doesn't need to be explicitly established between pvmds.

The flags bits are defined as follows:

SOM, EOM – Mark the first and last fragments (packets) of a message. Intervening fragments have both bits cleared. These are used by tasks and pvmd to detect message boundaries. When the pvmd refragments a packet in order to send it over a network with a small MTU, it adjusts the SOM and EOM bits as necessary.

DAT – Means that data is contained in the packet and the sequence number is

```
 Byte 0          1              2               3
+---------------+---------------+---------------+---------------+
|                          Message Code                         |
+---------------+---------------+---------------+---------------+
|      Message Encoding  -or-  Remote Wait Context Number       |
+---------------+---------------+---------------+---------------+
/                                                               /
```

Figure 14: Message Header

valid. The packet, even if zero-length, should be delivered.

ACK – Means that the acknowledgement number field is valid. This bit may be combined with the DAT bit to piggyback an acknowledgement on a data packet. Currently, however, the pvmd generates an acknowledgement packet for each data packet as soon as it is received, in order to get more accurate round-trip timing data.

FIN – Signals that the pvmd is closing down the connection. A packet with the FIN bit set (and DAT cleared) signals the first phase of an orderly shutdown. When an acknowledgement arrives (ACK bit and ack number matching the sequence number from the FIN packet), a final packet is sent with both FIN and ACK bits set. If the pvmd panics, (for example on a trapped segment violation) it tries to send a packet with FIN and ACK bits set to every peer before it exits.

The state of a connection between pvmds is kept in the host table entry (`struct hostd`). The state of a packet is kept in its `struct pkt`. Packets waiting to be sent to a host are queued in FIFO `hd_txq`. Packets may originate in local tasks or the pvmd itself, and are appended to this queue by the routing code. No receive queues are used, because incoming packets are passed immediately through to other send queues or reassembled into messages (or discarded). When the message is fully reassembled, the pvmd passes it to function `netentry()`, which dispatches it to the appropriate entry point. Figure 16 shows a diagram of packet routing inside the pvmd.

To improve performance over high-latency networks, the protocol allows multiple outstanding packets on a connection, so two more queues are required. `hd_opq` (and global `opq`) hold lists of unacknowledged packets. `hd_rxq` holds packets received out of sequence until they can be accepted.

When it arrives at the destination pvmd, each packet generates an acknowledgement packet back to the sender. The difference in time between sending a packet and getting the acknowledgement back is used to estimate the round-trip time to the foreign host. Each update is filtered into the estimate according to formula: $hd\_rtt_n = 0.75 * hd\_rtt_{n-1} + 0.25 * rtt$. When the acknowledgement for a packet comes back, it is removed from `hd_opq` and discarded. Each unacknowledged packet has a retry timer and count, and is resent until it is acknowledged by the foreign pvmd. The timer starts at three times the estimated round-trip time, and doubles for each retry until it reaches 18 seconds. The round-trip time estimate is limited to nine seconds and the backoff is bounded in order to allow at least 10 packets to be sent to a host before giving up. After three minutes of resending with no acknowledgement, a packets gets expired.

If a packet expires due to timeout, the foreign host or pvmd is assumed to be down or unreachable, and the local pvmd gives up on it (forever), calling `hostfailentry()`

All the parameters and default values mentioned above are defined in file `ddpro.h`.

10.4.2. Pvmd-Task Communication

A task talks to its pvmd over a TCP connection. UDP might seem more appropriate, as it is already a packet delivery service, whereas TCP is a stream protocol, requiring us to recreate packet boundaries. Unfortunately UDP isn't reliable; it can lose packets even within a host. Since an unreliable delivery system requires a retry mechanism (with timers) at both ends, and because one design assumption is that tasks can't be interrupted while computing to perform I/O, we're forced to use TCP. Note: We originally used UNIX-domain datagrams (analogous to UDP but used within a single host) for the pvmd-task connection. While this appeared to be reliable, it depends on the operating system implementation. More importantly, this protocol isn't as widely available as TCP.

10.4.3. Pvmd-Task Protocol

The packet delivery system between a pvmd and task is much simpler than between two pvmds because TCP offers reliable delivery. The pvmd and task maintain a FIFO of packets destined for each other, and switch between reading and writing on the TCP connection.

The main drawback with using TCP (as opposed to UDP) for the pvmd-task link is that the number of system calls needed to transfer a packet between a task and pvmd increases. Over UDP, a single `sendto()` and `recvfrom()` are required to transfer a packet. Since TCP provides no record marks (to distinguish back-to-back packets from one another), we have to send the overall packet length along with the header. So a packet can still be sent by a single `write()` call but, when done naively must be received by two `read()` calls, the first to get the header and the second to get the data.

When there is a lot of traffic on the pvmd-task connection, a simple optimization can reduce the average number of read calls back to about one per packet. If, when reading the packet body, the requested length of the *read* is increased by the size of a packet header, it may succeed in getting both the body of current packet and and header of the next packet at once. We have the header for the next packet for free and can read the body with a single call to *read*, so the average number of calls is reduced. Note: This was once implemented, but was removed while updating the code and hasn't yet been reintroduced.

The packet header is shown in Figure 15. No sequence numbers are needed, and the only flags are SOM and EOM, which are used as in the pvmd-pvmd protocol.

10.4.4. Databufs

The pvmd and libpvm both need to manage large amounts of dynamic data, mainly fragments of message text, often in multiple copies. In order to avoid copying, data is

```
 Byte 0         1              2              3
+---------------+--------------+--------------+---------------+
|                       Destination TID                       |
+---------------+--------------+--------------+---------------+
|                         Source TID                          |
+---------------+--------------+--------------+---------------+
|                        Packet Length                        |
+---------------+--------------+--------------+---------------+
| | | | | | |E|S|                                             |
| | | | | | |O|O|                 Unused                      |
| | | | | | |M|M|                                             |
+---------------+--------------+--------------+---------------+
/                                                             /
```

Figure 15: Pvmd-Task Packet Header

refcounted, allocating a few extra bytes for an integer at the head of the data. A pointer to the data itself is passed around, and the refcount maintenance routines subtract from this pointer to access the refcount or free the block. When the refcount of a databuf is decremented to zero, the block is freed.

10.4.5. Message Fragment Descriptors

PVM features dynamic-length messages, which means that a message can be composed without declaring its maximum length ahead of time. The pack functions allocate memory in steps, using databufs to store the data, and frag descriptors to chain the databufs together. Fragments are generally allocated with length equal to the largest UDP packet sendable by the pvmd. Space is reserved at the beginning of each fragment buffer to allow writing message and packet headers in place before sending. The `struct frag` used to keep fragments is defined in frag.h:

```
struct frag {
    struct frag *fr_link;   // chain or 0
    struct frag *fr_rlink;
    char *fr_buf;           // buffer or zero if master frag
    char *fr_dat;           // data
    int fr_max;             // size of buffer
    int fr_len;             // length of data
    struct {
        int ref : 16;       // refcount (of chain if master else of frag)
        int dab : 1;        // buffer is a databuf
        int spr : 1;        // sparse data (csz, lnc valid)
    } fr_u;
    int fr_csz;             // chunk size
    int fr_lnc;             // lead to next chunk
}
```

A frag holds a pointer (`fr_dat`) to a strip of data in memory and its length (`fr_len`).

It also keeps a pointer (`fr_buf`) to the allocated buffer containing the strip, and the length of the whole buffer (`fr_max`); these are used to reserve space to prepend or append data. A frag has forward and backward link pointers so it can be chained into a list; this is how a message is stored.

Each frag keeps a count of active references to it. When the refcount of a frag is decremented to zero, the frag descriptor is freed (and the underlying data refcount decremented). In the case where a frag descriptor is the head of a list, its refcount field applies to the entire list. When it reaches zero, every frag in the list is freed.

10.4.6. Packet Buffers

Packet descriptors are used to track message fragments inside the pvmd. Their structure is defined as follows:

```
struct pkt {
    struct pkt *pk_link;        // queue or 0
    struct pkt *pk_rlink;
    struct pkt *pk_tlink;       // scheduling queue or 0
    struct pkt *pk_trlink;
    int pk_src;                 // source tid
    int pk_dst;                 // dest tid
    int pk_flag;                // flags
    char *pk_buf;               // buffer or zero if master pkt
    int pk_max;                 // size of buffer
    char *pk_dat;               // data
    int pk_len;                 // length of data
    struct hostd *pk_hostd;     // receiving host
    int pk_seq;                 // seq num
    int pk_ack;                 // ack num
    struct timeval pk_rtv;      // time to retry
    struct timeval pk_rta;      // next-retry accumulator
    struct timeval pk_rto;      // total time spent on pkt
    struct timeval pk_at;       // time pkt first sent
    int pk_nrt;                 // retry count
}
```

The fields `pk_buf`, `pk_max`, `pk_dat` and `pk_len` are used in the same ways as the similarly named fields of a frag. The additional fields to track sparse data are not needed.

Unlike a frag, a packet can only be referenced in one place, so it doesn't have a refcount. The underlying data may be multiply referenced, though. In addition to data, pkts contain several fields necessary for operation of the pvmd-pvmd protocol. The pvmd-task protocol is much simpler, so the timer and sequence number fields are unused in pkts queued for tasks.

In function `netinput()` in the pvmd, packets are received directly into a packet buffer long enough to hold the largest packet the pvmd can receive. To route a packet, the pvmd simply chains it onto the end of the send queue for its destination. If the packet has multiple destinations (see multicasting section), the packet descriptor is replicated, counting extra references on the underlying databuf. After the last copy of the packet is sent, the databuf is freed.

In some cases, the pvmd can receive a packet (from a task) that is too long for the network interface of the destination host, or even the local pvmd. It refragments the packet by replicating the packet descriptor (similar to above). The `pk_dat` and `pk_len` fields of the descriptors are adjusted to cover successive chunks of the original packet, with each chunk small enough to send. At send time, in `netoutput()`, the pvmd saves under where it writes the packet header, sends the packet, then restores the data.

10.4.7. Message Buffers

In comparison to libpvm, the message packing functions in the pvmd are very simple. The message encoders/decoders handle only integers and strings. Integers occupy four bytes each with bytes in network order (bits 31..24 followed by bits 23..16, ...). Byte strings are packed as an integer length (including the terminating null if ASCII strings), followed by the bytes and zero to three bytes of zero to round the total length to a multiple of four. In libpvm, the "foo" encoder vector is used when talking to the pvmd. This encoding suffices for the needs of the pvmd, which never needs to pass around floating-point numbers or long/short integers.

In the pvmd as in libpvm, a message is stored in frag buffers, and can grow dynamically as more data is packed into it. The structure used to hold a message is:

```
struct mesg {
    struct mesg *m_link;    // chain or 0
    struct mesg *m_rlink;
    int m_ref;              // refcount
    int m_len;              // total length
    int m_dst;              // dst addr
    int m_src;              // src addr
    int m_enc;              // data encoding (for pvmd-task)
    int m_cod;              // type code
    int m_wid;              // wait serial (for pvmd-pvmd)
    int m_flag;
    struct frag *m_frag;    // master frag or 0 if we're master mesg
    struct frag *m_cfrag;   // keeps unpack state
    int m_cpos;             // keeps unpack state
};
```

10.4.8. Messages in the Pvmd

Functions `pkint()` and `pkstr()` append integers and null-terminated strings, respectively, onto a message. The corresponding unpacking functions are `upkint()` and `upkstr()`. Unsigned integers are packed as signed ones, but are unpacked using `upkuint()`. Another function, `upkstralloc()`, dynamically allocates space for the string it unpacks. All these functions use lower-level functions `bytepk()` and `byteupk()`, to write and read raw bytes to and from messages.

Messages are sent by calling function `sendmessage()`, which routes the message by its destination address. If for a remote destination, message fragments are attached to packets and delivered by the packet routing layer. If the message is addressed to the pvmd itself, `sendmessage()` simply passes the whole message descriptor to `netentry()`, the network message entry point, avoiding the overhead of the packet layer. This loopback interface is used often by the pvmd. For example, if it schedules a request and chooses itself as the target, it doesn't have to treat the message differently. It sends the message as usual and waits for a reply, which comes immediately. During a complex operation, `netentry()` may be reentered several times as the pvmd sends itself messages. Eventually the stack is unwound and a reply goes to the originator.

When it packetizes a message, `sendmessage()` prepends a message header (shown in Figure 14) to the first fragment before handing it off. The pvmd and libpvm use the same header for messages. *Code* contains an integer tag (message type). The second field has different interpretations to the pvmd and libpvm. Pvmds use the second field to pass the wait ID (if any, zero if none) associated with the message (operation). The usage of wait IDs was described earlier. Libpvm uses the second field to pass the encoding style of the message, as it can pack messages in a number of formats. When sending to another pvmd, `sendmessage()` sets the second field to `m_wid`, and when sending to a task, sets it to `m_cod` (1, or "foo").

Incoming messages are reassembled from packets by `loclinpkt()` if from a task or by `netinpkt()` if from another pvmd. Once reassembled, the appropriate entry point is called (`loclentry()`, `netentry()` or `schedentry()`). Using the tag in the message header, these functions multiplex control to one of the *dm_xxx*(), *tm_xxx*() or *sm_xxx*() entry points if the tag has a legal value, otherwise the message is discarded. Each of the entry points performs a specific function in the pvmd, In general it unpacks parameters from the message body, takes some action (or looks up some data), and generates a response message.

Pvmds take almost no autonomous action, rather syscalls initiated by tasks are what cause things to happen. The only functions that pvmds do automatically are to ping other pvmds to check network health and delete down hosts from the machine configuration.

A graph of packet and message routing inside the pvmd is shown in Figure 16.

10.4.9. Message Encoders

To allow the PVM programmer to manage message buffers, for example to save, recall or get information about them, they are labeled with integer message IDs (MIDs). Each message buffer has a unique MID, which is its index in the message heap, allowing it to be located quickly. When a message buffer is freed, its MID is recycled. The message heap starts out small and is extended as it runs out of free MIDs.

Libpvm provides a set of functions for packing typed data into messages and recovering it at the other end. Any primitive data type can be packed into a message, in one of several encoding formats. Each message buffer holds a vector of functions for encoding/decoding all the primitive types (`struct encvec`), initialized when the buffer is created. So, for example, to pack a long integer the generic pack function `pvm_pklong()` calls `(ub_codef->enc_long)()` of the current pack buffer.

There are currently five sets of encoders (and decoders) defined. The encoder/decoder

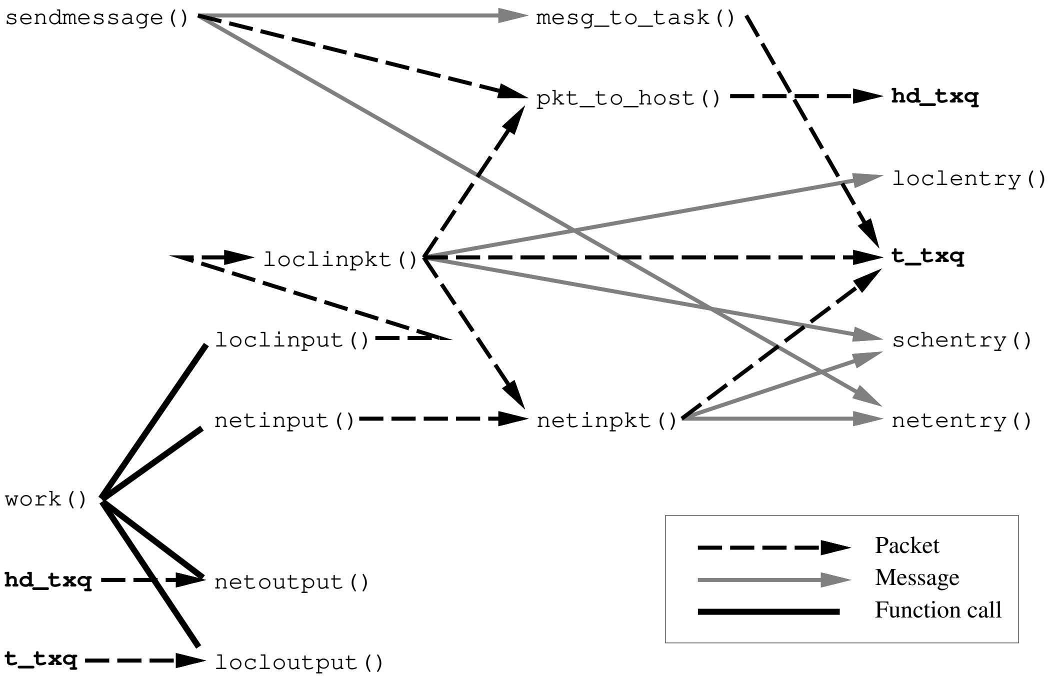

Figure 16: Packet and Message Routing

vector used in a buffer is determined by the format parameter passed to `pvm_mkbuf()` when creating a new message, and by the encoding field of the message header when receiving a message. The two most commonly used ones pack data into "raw" (host native) and "default" (XDR) formats. "Inplace" encoders pack descriptors of the data instead of the data itself. The data is left in place until the message is actually sent. There are no inplace decoders; these entries call a function that always returns an error. "Foo" encoders can pack only integers and strings, and must be used when composing a message for the pvmd. Finally, "alien" decoders are installed when a received message can't be unpacked because its format doesn't match any of the decoders available in the task. This allows the message to be held or forwarded, but any attempt to read data from it will result in an error.

One drawback to using encoder vectors is that, since they "touch" every function for every format, the linker must include all the functions out of libpvm in every executable, even if they're not used.

10.4.10. Packet Handling Functions

Four functions handle all packet traffic into and out of libpvm.

`Mroute()` is called by higher-level functions such as `pvm_send()` and `pvm_recv()` to send and receive messages. It establishes any necessary routes before calling `mxfer()`.

`Mxfer()` polls for messages, possibly blocking until one is received or until a specified timeout. It calls `mxinput()` to copy fragments into the task and assemble them into messages. In the generic version of PVM, `mxfer()` uses `select()` to poll all routes (sockets) in order to find those ready for input or output.

`Pvmmctl()` is called by `mxinput()` whenever a control message is received. Control messages are covered in the next section.

10.4.11. Control Messages

Control messages are sent like regular messages to a task, but have tags in a reserved space (between `TC_FIRST` and `TC_LAST`). When the task downloads a control message, instead of queueing it for receipt, it passes the message to the `pvmmctl()` function, and then discards it. Like `loclentry()` in the pvmd, `pvmmctl()` is an entry point in the task, causing it to take some action. The main difference is that control messages can't always be used to get the task's attention, since it must be in `mxfer()`, sending or receiving in order to get them.

The following control message tags are defined. The first three are used by the direct routing mechanism which is discussed in the next section. In the future control messages may be used to do things such as set debugging and tracing masks in the task as it runs.

Tag	Meaning
TC_CONREQ	Connection request
TC_CONACK	Connection ack
TC_TASKEXIT	Task exited/doesn't exist
TC_NOOP	Do nothing
TC_OUTPUT	Claim child stdout data

10.4.12. Message Direct Routing

Direct routing allows one task to send messages to another through a TCP link, avoiding the overhead of copying them through the pvmds. This mechanism is implemented entirely in libpvm, by taking advantage of the notify and control message facilities.

By default, any message sent to another task is routed to the pvmd, which forwards it to the destination. If direct routing is enabled ($pvmrouteopt = PvmRouteDirect$) when a message (addressed to a task) is passed to `mroute()`, it attempts to create a direct route if one doesn't already exist. The route may be granted or refused by the destination task, or fail (if the destination doesn't exist). The message and route (or default route) are then passed to `mxfer()`.

Libpvm maintains a protocol control block (`struct ttpcb`) for each active or denied connection, in list `ttlist`. To request a connection, `mroute()` makes a new ttpcb and creates and binds a socket. It sends a TC_CONREQ control message to the destination via the default route. At the same time, it sends a TM_NOTIFY message to the pvmd, to be notified if the destination task exits, with closure (message tag) TC_TASKEXIT. Then it puts the ttpcb in TTCONWAIT state, and waits until the state of the ttpcb changes to something other than TTCONWAIT, calling `mxfer()` in blocking mode repeatedly to receive messages.

When the destination task enters `mxfer()`, for example to receive a message, it gets the TC_CONREQ message. If its routing policy ($pvmrouteopt! = PvmDontRoute$) and libpvm implementation allow a direct connection, and it has resources available, and the protocol version (TDPROTOCOL) in the request matches its own, it grants the request. It makes a ttpcb with state TTGRNWAIT, creates and binds a socket and listens on it, then replies with a TC_CONACK message. If the destination denies the connection, it creates a ttpcb with state TTDENY and nacks with a TC_CONACK message. The originator receives the TC_CONACK message, and either opens the connection ($state = TTOPEN$) or marks the route denied ($state = TTDENY$). Finally, `mroute()` passes the original message to `mxfer()`, which sends it. Denied connections must be cached in order to prevent repeated negotiation.

If the destination doesn't exist, the TC_CONACK message never arrives because the TC_CONREQ message is silently dropped by the pvmds. However, the TC_TASKEXIT message generated by the notify system arrives in its place, and the ttpcb state is set to TTDENY.

This connect scheme also works if both ends try to establish a connection at the same time. They both enter TTCONWAIT, and when they receive each others' TC_CONREQ messages, they go directly to the TTOPEN state. The state diagram for a connection is shown in Figure 17.

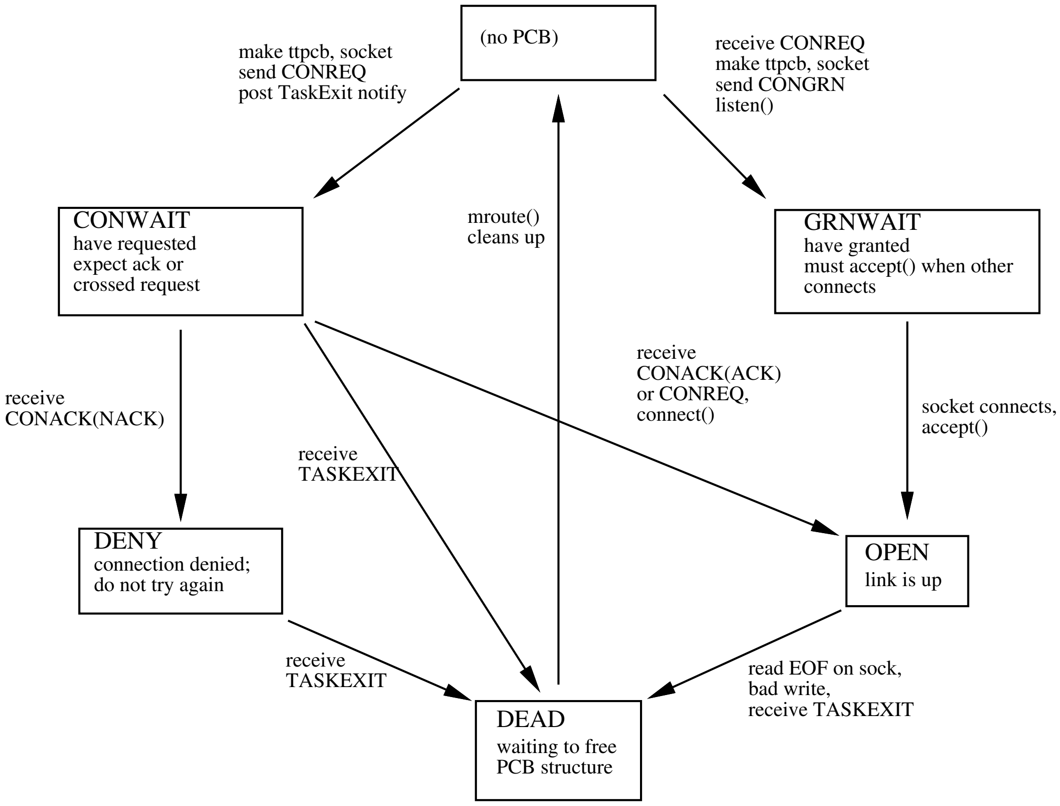

Figure 17: Task-Task Connection State Diagram

10.4.13. Multicasting

Libpvm provides a function, `pvm_mcast()`, that sends a message to multiple destinations simultaneously, hopefully in less time than several calls to `pvm_send()`. The current implementation only routes multicast messages through the pvmds and uses a 1:N fanout to simplify the fault-tolerance issues. The problem is to ensure that failure of a host doesn't cause the loss of any messages (other than ones to that host). The packet routing layer of the pvmd cooperates with the libpvm to multicast a message.

To form a multicast address TID (or GID), the G bit is set (refer to Figure 10). Each pvmd owns part of the GID-space, with the H field set to its host index (as with TIDs). The L field is assigned by a counter that is incremented for each multicast. So, a new multicast address is used for each message, then recycled. The pvmd uses a `struct mca` to keep a record of active multicasts.

To initiate a multicast, the task sends a TM_MCA message to its pvmd, containing a list of all recipient tids. In `tm_mca()`, the pvmd creates a new multicast descriptor and GID. It sorts them, removes bogus ones and duplicates and caches the list of addresses in the mca. Next, to each destination pvmd in the multicast list (ones with destination tasks), it sends a DM_MCA message containing the destinations on that host. Finally, the GID is sent back to the task in the TM_MCA reply message.

The task now sends the multicast message to the pvmd, addressed to the multicast address. As each packet arrives at the pvmd, the routing layer replicates it once for each local destination (tasks on the same host), and once for each foreign pvmd. When a multicast packet arrives at a destination pvmd, it is again replicated and delivered to each destination task. The pvmd-pvmd communication preserves packet order, so the multicast address and data packets arrive in order at each destination.

As it forwards multicast packets, each pvmd eavesdrops on the header flags. When it sees a packet with bit EOM set, the pvmd knows it has reached the end of the multicast message, and flushes the mca.

10.5. Environment Variables

Experience seems to indicate that inherited environment (UNIX `environ`) is useful to an application. For example, environment variables can be used to distinguish a group of related tasks or set debugging variables.

PVM makes increasing use of environment, and will probably eventually support it even on machines where the concept is not native. For now, it allows a task to export any part of `environ` to tasks spawned by it. Setting variable `PVM_EXPORT` to the names of other variables causes them to be exported through spawn. For example, setting:

```
PVM_EXPORT=DISPLAY:SHELL
```

exports the variables `DISPLAY` and `SHELL` to children tasks (and `PVM_EXPORT` too).

10.6. Standard Input and Output

Each task, except for anonymous ones (not started by *spawn*) inherits a *stdout* sink from its parent. Any output generated by the task is sent to this device, packed into PVM messages. The sink is a $< TID, code >$ pair; messages are sent to the TID with tag equal to the specified code. The tag helps the message sink task select messages to receive and identify the source (since it may have no prior knowledge of the task from which the message originates).

Output messages for a task come from its pvmd, since it reads the pipe connected to the task's stdout. If the output TID is set to zero (the default for a task with no parent), the messages go to the master pvmd, where they are written on its error log.

Children spawned by a task inherit its output sink. Before the spawn, the parent can use `pvm_setopt` to alter the output TID or code. This doesn't affect where the output of the parent task itself goes. A task may set output-TID to one of three things: The value inherited from its parent, its own TID or zero. It can set output-code only if outputTID is set to its own TID. This means that output can't be assigned to an arbitrary task. It's not clear this restriction is a good one.

Four types of messages are sent to an output sink. The message body formats for each type are:

```
Spawn:
(code) {                    // task has been spawned
        int tid,            // task id
        int -1,             // signals spawn
        int ptid            // TID of parent
}

Begin:
(code) {                    // first output from task
        int tid,            // task id
        int -2,             // signals task creation
        int ptid            // TID of parent
}

Output:
(code) {                    // output from a task
        int tid,            // task id
        int count,          // length of output fragment
        char data[count]    // output fragment
}

EOF:
(code) {                    // last output from a task
        int tid,            // task id
        int 0               // signals EOF
}
```

The first two items in the message body are always the task ID and output count, which distinguishes between the four message types. For each task, one message each with count equal to -1, -2, and 0 will be sent, along with zero or more messages with count > 0. Types -2, > 0 and 0 will be received in order, as they originate from the same source (the pvmd of the target task). Type -1 originates at the pvmd of the parent task, so it can be received in any order relative to the others.

The output sink is expected to understand the different types of messages and use them to know when to stop listening for output from a task (EOF) or group of tasks (global EOF). The messages are designed this way to prevent race conditions when a task spawns another task, then immediately exits. The output sink might get the EOF message from the first task and decide the group is finished, only to find more output later from second task. But either the -2 message or the -1 message for the second task must arrive before the 0 message from the first task. The states of a task as inferred from output messages received are shown in Figure 18.

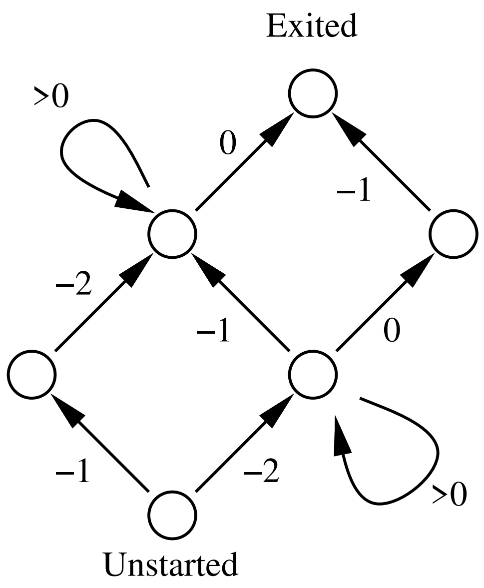

Figure 18: Output States of a Task

The libpvm function `pvm_catchout()` uses this output collection feature to put the output from children of a task into a file (for example its own stdout stream).

It sets output TID to its own task ID, and the output code to `TC_OUTPUT`, which is a control message. Output from children and grandchildren tasks is collected by the pvmds and sent to the task, where it is received by `pvmmctl()` and printed by `pvmclaimo()`.

10.7. Tracing

PVM includes a task tracing system built into the libpvm library, which records the parameters and results of all calls to libpvm functions. This description is sketchy because this is the release of the tracing code.

Libpvm generates trace-event messages when any of the functions is called, and sends the messages to its inherited trace data sink. Tasks also inherit a trace mask, which is used to enable tracing per function. The mask is passed as a hexadecimal string in environment variable `PVMTMASK`. Trace data isn't generated at all if tracing isn't enabled (since it's expensive).

Constants related to interpreting trace messages are defined in public header file `pvmtev.h`. Trace data from a task is collected in a manner similar to the output redirection discussed above. Like the type -1, -2 and 0 messages which bracket output from a task, `TEV_SPNTASK`, `TEV_NEWTASK` and `TEV_ENDTASK` trace messages are generated by the pvmds to bracket trace messages.

10.8. Console Internals

The PVM console is used to manage the virtual machine – to reconfigure it or start and stop processes. In addition, it's an example program that makes use of most of the libpvm functions.

The console uses `pvm_getfds()` and `select()` to check for input from the keyboard and messages from the pvmd simultaneously. Keyboard input is passed to the command interpreter, while messages contain notification (for example HostAdd) or output from a task.

The console can use output redirection (described earlier) to collect output from spawned tasks. Normally, when spawning a task the console sets output-TID to zero, so any output goes to the default sink (for example, the master pvmd log file). Using spawn flags `->` or `->>` causes the console to set output-TID to itself and output-code to a unique "job" number (assigned by a counter).

Unless some intermediate task redirects the output again, when output is generated by child tasks or their children, it is sent back to the console. By assigning a unique code to each task spawned, the console can maintain separate "jobs" or "process groups", which are sets of tasks with matching output codes. Most of the code to handle output redirection is in the console, while only a few small changes were made to the pvmd and libpvm. We chose this route because it keeps the complexity out of the core of the system.

The console has a `tickle` command, which in turn calls libpvm function `pvm_tickle()`. This is used to set the pvmd debug mask and dump the contents of various data structures. For example, the command `tickle 6 18` sets the pvmd debug mask to 0x18 (bits 3 and 4) and `tickle 1` dumps the current host table (to the pvmd log file). `pvm_tickle()` is an undocumented function in libpvm and not considered an official part of the PVM interface. Nevertheless, if you wish to use this function, the options for tickle can be found by typing `help tickle` in the console.

10.9. Resource Limitations

Resource limits imposed by the operating system and available hardware are in turn passed to PVM applications. Whenever possible, PVM tries to avoid setting explicit limits, rather it returns an error when resources are exhausted. Naturally, competition between users on the same host or network affects some limits dynamically.

10.9.1. In the PVM Daemon

How many tasks each pvmd can manage is limited by two factors: The number of processes allowed a user by the operating system, and the number of file descriptors available to the pvmd. The limit on processes is generally not an issue, since it doesn't make sense to have a huge number of tasks running on a uniprocessor machine.

Each task consumes one file descriptor in the pvmd, for the pvmd-task TCP stream. Each spawned task (not ones connected anonymously) consumes an extra descriptor, since its output is read through a pipe by the pvmd (closing stdout and stderr in the task would reclaim this slot). A few more file descriptors are always in use by the pvmd for the local and network sockets and error log file. For example, with a limit of 64 open files, a user should be able to have up to 30 tasks running per host.

The pvmd may become a bottleneck if all these tasks try to talk to one another through it.

The pvmd uses dynamically allocated memory to store message packets en route between tasks. Until the receiving task accepts the packets, they accumulate in the pvmd in a FIFO. No flow control is imposed by the pvmd – it will happily store all the packets given to it, until it can't get any more memory. If an application is designed so that tasks can keep sending even when the receiving end is off doing something else and not receiving, the system will eventually run out of memory.

10.9.2. In the Task

As with the pvmd, a task may have a limit on the number of others it can connect to directly. Each direct route to a task has a separate TCP connection (which is bidirectional), and so consumes a file descriptor. Thus with a limit of 64 open files, a task can establish direct routes to about 60 other tasks. Note this limit is only in effect when using task-task direct routing. Messages routed via the pvmds only use the default pvmd-task connection.

The maximum size of a PVM message is limited by the amount of memory available to the task. Because messages are generally packed using data existing elsewhere in memory, and they must reside in memory between being packed and sent, the largest possible message a task can send should be somewhat less than half the available memory. Note that as a message is sent, memory for packet buffers is allocated by the pvmd, aggravating the situation. Inplace message encoding alleviates this problem somewhat, because the data is not copied into message buffers in the sender. However, on the receiving end, the entire message is downloaded into the task before the receive call accepts it, possibly leaving no room to unpack it.

In a similar vein, if many tasks send to a single destination all at once, the destination task or pvmd may be overloaded as it tries to store the messages. Keeping messages from being freed when new ones are received by using `pvm_setrbuf()` also uses up memory.

These problems can sometimes be avoided by rearranging the application code, for example to use smaller messages, eliminate bottlenecks, and process messages in the order in which they are generated.

10.10. Multiprocessor Ports

This section describes the technical details of the PVM multiprocessor ports to message-passing multicomputers as well as shared-memory systems. The implementations and related issues are discussed to assist the experienced programmers who are interested in porting PVM to other multiprocessor platforms.

PVM provides an interface that hides the system details from the programmer. PVM applications will run unchanged between multicomputer and workstations as long as file I/O and the multicomputer's memory limitations are respected. The only thing that needs to be changed is the Makefile. The user does not have to know how to allocate nodes on the system or how to load a program onto the nodes, since PVM takes care of these tasks.

A single PVM daemon runs on the iPSC/860, CM-5, and T3D MPP systems and serves as the gateway to the outside world. On some sytems this requires the pvmd be run on a front-end machine and to be built with a different compiler. On other MPP systems such as the Paragon and the IBM SP-2 one pvmd runs on each computational node. On most shared-memory systems the operating system selects a processor to run the pvmd, and may even migrate the pvmd.

Because the Paragon OS creates proxy processes when executing scripts, it is generally not possible to "add" the Paragon to a virtual machine. Instead, the user should start PVM on the Paragon and then "add" outside hosts. For example, to start PVM on a four node partition type:

```
pexec $PVM_ROOT/lib/PGON/pvmd3 -sz 4 &
pvm
```

At this point the user can add other hosts or run a PVM application.

Note that a useful hack for Paragon sites running PVM is to modify the PVM_ROOT/lib/pvmd script to account for the fact that the PVM daemon starts in the compute partition. To keep the PVM daemon from trying to grab the entire compute partition, the penultimate line of this script can be modified to something like:

```
exec $PVM_ROOT/lib/$PVM_ARCH/pvmd3 -pn `whoami` $@
```

This hack forces a Paragon user to create a specifically named partition to run PVM in; if the partition does not exist then the daemon startup will fail. Such local modifications to the Paragon pvmd script can be done on a site-wide or per-user basis to suit the needs of PVM users or the Paragon system administrator.

10.10.1. Message Passing Architectures

On MPPs where message-passing is supported by the operating system, the PVM message-passing functions are translated into the native send and receive system calls. Since the TID contains the task's location, the messages to be sent directly to the target task, without any help from the daemon.

When a task calls `pvm_spawn()`, the daemon handles the request and loads the new processes onto the nodes. The way PVM allocates nodes is system-dependent. On the CM5, the entire partition is allocated to the user when he logs on. On the iPSC/860, PVM will get a subcube big enough to accommodate all the tasks to be spawned; only tasks spawned together reside in the same subcube. (Note the NX operating system limits the number of active subcubes system wide to 10. pvm_spawn() will fail when this limit is reached or when there are not enough nodes available.) In the case of the

Paragon, PVM uses the default partition unless a different one is specified when pvmd is invoked. Pvmd and the spawned tasks form one giant parallel application. The user can set the appropriate NX environment variables such as NX_DFLT_SIZE before starting PVM, or he can specify the equivalent command-line arguments to pvmd (i.e., pvmd -sz 32).

PVM uses the native *asynchronous* message-passing primitives whenever possible. One drawback to this choice is that the operating system can run out of message handles or buffer space very quickly if a lot of messages are sent at once. In this case, PVM will be forced to switch to synchronous send. To improve performance, a task should call pvm_send() as soon as the data become available, so (hopefully) when the other task calls pvm_recv() the message will already be in its buffer. PVM buffers one incoming packet between calls to pvm_send()/pvm_recv(). A large message, however, is broken up into many fixed-size fragments during packing, and each piece is sent separately. The size of these fragments is set by MAXFRAGSIZE in `pvmmimd.h`. Buffering one of these fragments won't do much good unless pvm_send() and pvm_recv() are synchronized.

10.10.2. Shared-Memory Architectures

In the shared-memory implementation, each task owns a shared buffer created with a shmget() (or equivalent) system call. The task ID is used as the "key" to the shared segment. A task communicates with other tasks by mapping their message buffers into its own memory space.

To enroll in PVM, the task first writes its UNIX process ID into pvmd's incoming box. It then looks for the assigned task ID in pvmd's pid→tid table.

The message buffer is divided into pages, each holds one fragment. The fragment size is therefore equal to the system page size subtracted by the size of the shared-memory header, which contains the lock and the reference count. The first page is the incoming box, while the rest of the pages hold outgoing fragments. To send a message, the task first packs the message body into its buffer, then delivers the message header, which contains the sender's TID and the location of the data, to the incoming box of the intended recipient. When pvm_recv() is called, PVM checks the incoming box, locates and unpacks the messages (if any), and decreases the reference count so the space can be reused. If a task is not able to deliver the header directly because the receiving box is full, it will block until the other task is ready.

Inevitably some overhead will be incurred when a message is packed into and unpacked from the buffer, as is the case with all other PVM implementations. If the buffer is full, then the data must first be copied into a temporary buffer in the process's private space and later transferred to the shared buffer.

Memory contention is usually not a problem. Each process has its own buffer and each page of the buffer has its own lock. Only the page being written to is locked, and no process should be trying to read from this page because the header has not been sent out. Different processes can read from the same page without interfering with each other, so multicasting will be efficient (they do have to decrease the counter afterwards, resulting in some contention). The only time contention occurs is when two or more processes trying to deliver the message header to the same process at the same time. But since the header is very short (8 bytes), such contention should not cause any significant delay.

To minimize the possibility of page faults, PVM attempts to use only a small number of pages in the message buffer and recycle them as soon as they have been read by all intended recipients.

Once a task's buffer has been mapped, it will not be unmapped, unless the system limits the number of mapped segments. This saves time for any subsequent message exchanges with the same process.

10.10.3. Functions to Port

Seven functions serve as the MPP "interface" for PVM. The implementation of these functions is system dependent, and the source code should be kept in the file `pvmdmimd.c` (message-passing) or `pvmdshmem.c` (shared-memory). We give a brief description of each of these functions below.

```
void mpp_init(int argc, char **argv);
    Initialization. Called once when PVM is started. Arguments argc and argv
    are passed from pvmd main().

int mpp_load(int flags, char *name, char *argv, int count, int *tids, int ptid);
    Create partition if necessary. Load executable onto nodes; create new
    entries in task table, encode node number and process type into task IDs.
    flags:  exec options;
    name:   executable to be loaded;
    argv:   command line argument for executable;
    count:  number of tasks to be created;
    tids:   array to store new task IDs;
    ptid:   parent task ID.

void mpp_output(struct task *tp, struct pkt *pp);
    Send all pending packets to nodes via native send. Node number and process
    type are extracted from task ID.
    tp: destination task;
    pp: packet.

int mpp_mcast(struct pkt pp, int *tids, int ntask);
    Global send.
    pp:     packet;
    tids:   list of destination task IDs;
    ntask:  how many.

int mpp_probe();
    Probe for pending packets from nodes (non-blocking). Returns 1 if packets
    are found, otherwise 0.

void mpp_input();
    Receive pending packets (from nodes) via native receive.

void mpp_free(int tid)
```

```
Remove node/process-type from active list.
tid: task ID.
```

10.11. Debugging the PVM Source

To help catch memory allocation errors in the system code, the pvmd and libpvm use a sanity-checking library called *imalloc*. Imalloc functions are wrappers for the regular *libc* functions `malloc()`, `realloc()` and `free()`. Upon detecting an error, the imalloc functions abort the program so the fault can be traced.

The following checks and functions are performed by imalloc:

1. The length argument to malloc is checked for insane values. A length of zero is changed to one so it succeeds.

2. Each allocated block is tracked in a hash table to detect when `free()` is called more than once on a block or on something not from `malloc()`.

3. `I_malloc()` and `i_realloc()` write pads filled with a pseudo-random pattern outside the bounds of each block, which are checked by `i_free()` to detect when something writes past the end of a block.

4. `I_free()` zeros each block before it frees it so further references may fail and make themselves known.

5. Each block is tagged with a serial number and string to indicate its use. The heap space can be dumped or sanity-checked at any time by calling `i_dump()`. This helps find memory leaks.

Since the overhead of this checking is quite severe, it is disabled at compile time by default. Defining `USE_PVM_ALLOC` in the source Makefile(s) switches it on.

The pvmd and libpvm each have a debugging mask that can be set to enable logging of various information. Logging information is divided up into classes, each of which is enabled separately by a bit in the debug mask. The pvmd command line option `-d` sets the debug mask of the pvmd to the (hexadecimal) value specified; the default is zero. Slave pvmds inherit the debug mask of the master at the time they are started. The debug mask of a pvmd can be set at any time using the console `tickle` command on that host. The debug mask in libpvm can be set in the task with `pvm_setopt()`.

Note: The debug mask is not intended for debugging application programs.

The pvmd debug mask bits are defined in ddpro.h, and the libpvm bits in lpvm.c. The meanings of the bits are not well defined and are subject to change, as they're intended to be used when fixing or modifying the pvmd or libpvm. Presently, the bits in the debug mask correspond to:

Name	bit	debug messages about
`pkt`	1	packet routing
`msg`	2	message routing
`tsk`	4	task creation/exit
`slv`	8	slave pvmd configuration
`hst`	10	host table updates
`sel`	20	select loop in pvmd (below packet routing layer)
`net`	40	network twiddling
`mpp`	80	mpp related options
`sch`	100	scheduler interface

The pvmd includes several registers and counters to sample certain events, such as the number of calls made to `select()` or the number of packets refragmented by the network code. These values can be computed from a debug log, but the counters have less adverse impact on the performance of the pvmd than would generating a huge log file. The counters can be dumped or reset using the `pvm_tickle()` function or the console tickle command. The code to gather statistics is normally switched out at compile-time. To enable it, edit the makefile and add `-DSTATISTICS` to the compile options.

11. Support

Several avenues exist for getting help with using PVM. A PVM bulletin board exists on the Internet for users to exchange ideas, tricks, successes and problems. The news group name is `comp.parallel.pvm`. Several vendors including Cray Research, Convex, SGI, IBM, Intel, DEC, and Thinking Machines have decided to supply and support PVM software on their systems. Several software companies have also sprung up to offer user installation and support for PVM. The PVM developers also answer mail as time permits: PVM problems or questions can be sent to `pvm@msr.epm.ornl.gov` for a quick and friendly reply. The first annual PVM User's Group meeting was held in Knoxville in May 1993. The slides from this meeting are available in postscript form by ftp from netlib2@cs.utk.edu in the pvm3/ug directory.

12. References

[1] Beguelin, Dongarra, Geist, Manchek, Sunderam A User's Guide to PVM (Parallel Virtual Machine) ORNL/TM-11826, July 1991.

[2] T. Green, J. Pasko DQS 2.x/3.0 Proceedings of Cluster Workshop '93 at SCRI Florida State University. Dec. 1993.

[3] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunder of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[4] R. Manchek PVM Design Master's Thesis University of Tennessee, June 1994.

[5] Platform Computing Corporation 203 College St. Suite 303. Toronto Ontario.

[6] B. Schmidt, V. Sunderam Empirical Analysis of Overheads in Cluster Environments Concurrency: Practice and Experience 6 (1), pp 1-32 February 1994.

13. Appendix A. Reference pages for PVM 3 routines

This appendix contains an alphabetical listing of all the PVM 3 routines. Each routine is described in detail for both C and Fortran use. There are examples and diagnostics for each routine.

pvmfaddhost() pvm_addhosts()

Adds one or more hosts to the virtual machine.

Synopsis

C	`int info = pvm_addhosts( char **hosts, int nhost, int *infos )`
Fortran	`call pvmfaddhost( host, info )`

Parameters

`hosts`	–	an array of pointers to character strings containing the names of the machines to be added.
`nhost`	–	integer specifying the number of hosts to be added.
`infos`	–	integer array of length `nhost` which contains the status code returned by the routine for the individual hosts. Values less than zero indicate an error.
`host`	–	character string containing the name of the machine to be added.
`info`	–	integer status code returned by the routine. Values less than nhost indicate partial failure, values less than 1 indicate total failure.

Discussion

The routine `pvm_addhosts` adds the list of computers pointed to in `hosts` to the existing configuration of computers making up the virtual machine. If pvm_addhosts is successful `info` will be equal to `nhost`. Partial success is indicated by 1 <=`info`<`nhost`, and total failure by `info`< 1. The array `infos` can be checked to determine which host caused the error.

The Fortran routine `pvmfaddhost` adds a single host to the configuration with each call.

If a host fails, the PVM system will continue to function. The user can use this routine to increase the fault tolerance of the PVM application. The status of hosts can be requested by the application using pvm_mstat and pvm_config. If a host has failed it will be automatically deleted from the configuration. Using pvm_addhosts a replacement host can be added by the application. It is still the responsibility of the application developer to make the application tolerant of host failure. Another use of this feature would be to add more hosts as they become available, for example on a weekend, or if the application dynamically determines it could use more computational power.

Examples

C:

```
static char *hosts[] = {
        "sparky",
        "thud.cs.utk.edu",
};
info = pvm_addhosts( hosts, 2, infos );
```

Fortran:

```
CALL PVMFADDHOST( 'azure', INFO )
```

Errors

The following error conditions can be returned by `pvm_addhosts`

Name	Possible cause
PvmBadParam	giving an invalid argument value.
PvmAlready	already been added.
PvmSysErr	local pvmd is not responding.

The following error conditions can be returned in `infos`

Name	Possible cause
PvmBadParam	bad hostname syntax.
PvmNoHost	no such host.
PvmCantStart	failed to start pvmd on host.
PvmDupHost	host already in configuration.
PvmBadVersion	remote pvmd version doesn't match.
PvmOutOfRes	PVM has run out of system resources.

pvmfbarrier() pvm_barrier()

Blocks the calling process until all processes in a group have called it.

Synopsis

C	`int info = pvm_barrier( char *group, int count )`
Fortran	`call pvmfbarrier( group, count, info )`

Parameters

- `group` – character string group name. The group must exist and the calling process must be a member of the group.
- `count` – integer specifying the number of group members that must call pvm_barrier before they are all released. Though not required, count is expected to be the total number of members of the specified group.
- `info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_barrier` blocks the calling process until `count` members of the `group` have called pvm_barrier. The count argument is required because processes could be joining the given group after other processes have called pvm_barrier. Thus PVM doesn't know how many group members to wait for at any given instant. Although count can be set less, it is typically the total number of members of the group. So the logical function of the pvm_barrier call is to provide a group synchronization. During any given barrier call all participating group members must call barrier with the same count value. Once a given barrier has been successfully passed, pvm_barrier can be called again by the same group using the same group name.

As a special case if `count` equals -1 then PVM will use the value of pvm_gsize() i.e. all the group members. This case is useful after a group is established and not changing during an application.

If pvm_barrier is successful, `info` will be 0. If some error occurs then `info` will be < 0.

Examples

C:

```
inum = pvm_joingroup( "worker" );
        .
        .
info = pvm_barrier( "worker", 5 );
```

Fortran:

```
CALL PVMFJOINGROUP( 'shakers', INUM )
COUNT = 10
CALL PVMFBARRIER( 'shakers', COUNT, INFO )
```

Errors

These error conditions can be returned by `pvm_barrier`

Name	Possible cause
PvmSysErr	pvmd was not started or has crashed.
PvmBadParam	giving a count < 1.
PvmNoGroup	giving a non-existent group name.
PvmNotInGroup	calling process is not in specified group.

pvmfbcast() pvm_bcast()

broadcasts the data in the active message buffer.

Synopsis

C	`int info = pvm_bcast( char *group, int msgtag )`
Fortran	`call pvmfbcast( group, msgtag, info )`

Parameters

`group`	–	character string group name of an existing group.
`msgtag`	–	integer message tag supplied by the user. msgtag should be >= 0. It allows the user's program to distinguish between different kinds of messages .
`info`	–	integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_bcast` broadcasts a message stored in the active send buffer to all the members of `group`. In PVM 3.2 the broadcast message is not sent back to the sender. Any PVM task can call pvm_bcast(), it need not be a member of the group. The content of the message can be distinguished by `msgtag`. If pvm_bcast is successful, `info` will be 0. If some error occurs then `info` will be < 0.

pvm_bcast is asynchronous. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processors. This is in contrast to synchronous communication, during which computation on the sending processor halts until a matching receive is executed by all the receiving processors.

pvm_bcast first determines the tids of the group members by checking a group data base. A multicast is performed to these tids. If the group is changed during a broadcast the change will not be reflected in the broadcast. Multicasting is not supported by most multiprocessor vendors. Typically their native calls only support broadcasting to *all* the user's processes on a multiprocessor. Because of this omission, pvm_bcast may not be an efficient communication method on some multiprocessors.

Examples

C:

```
info = pvm_initsend( PvmDataRaw );
info = pvm_pkint( array, 10, 1 );
msgtag = 5 ;
info = pvm_bcast( "worker", msgtag );
```

Fortran:

```
CALL PVMFINITSEND( PVMDEFAULT )
CALL PVMFPKFLOAT( DATA, 100, 1, INFO )
CALL PVMFBCAST( 'worker', 5, INFO )
```

Errors

These error conditions can be returned by `pvm_bcast`

Name	Possible cause
PvmSysErr	pvmd was not started or has crashed.
PvmBadParam	giving a negative msgtag.
PvmNoGroup	giving a non-existent group name.

pvmfbufinfo() pvm_bufinfo()

returns information about the requested message buffer.

Synopsis

C
```
int info = pvm_bufinfo( int bufid, int *bytes,
                        int *msgtag, int *tid )
```
Fortran
```
call pvmfbufinfo( bufid, bytes, msgtag, tid, info )
```

Parameters

`bufid`	–	integer specifying a particular message buffer identifier.
`bytes`	–	integer returning the length in bytes of the entire message.
`msgtag`	–	integer returning the message label.
`tid`	–	integer returning the source of the message.
`info`	–	integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_bufinfo` returns information about the requested message buffer. Typically it is used to determine facts about the last received message such as its size or source. pvm_bufinfo is especially useful when an application is able to receive any incoming message, and the action taken depends on the source `tid` and the `msgtag` associated with the message that comes in first. If pvm_bufinfo is successful, `info` will be 0. If some error occurs then `info` will be < 0.

Examples

C:
```
bufid = pvm_recv( -1, -1 );
info = pvm_bufinfo( bufid, &bytes, &type, &source );
```
Fortran:
```
CALL PVMFRECV( -1, -1, BUFID )
CALL PVMFBUFINFO( BUFID, BYTES, TYPE, SOURCE, INFO )
```

Errors

These error conditions can be returned by `pvm_bufinfo`:

Name	Possible cause
PvmNoSuchBuf	specified buffer does not exist.
PvmBadParam	invalid argument

pvmfcatchout() pvm_catchout()

Catch output from child tasks.

Synopsis

C	`#include <stdio.h>`
	`int bufid = pvm_catchout( FILE *ff )`
Fortran	`call pvmfcatchout( onoff )`

Parameters

`ff`	–	File descriptor on which to write collected output.
`onoff`	–	Integer parameter. Turns output collection on or off.

Discussion

The routine `pvm_catchout` causes the calling task (the parent) to catch output from tasks spawned after the call to pvm_catchout. 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. Output from grandchildren (spawned by children) tasks is also collected, provided the children don't reset PvmOutputTid using pvm_setopt().

Each line of output has one of the following forms:

```
[txxxxx] BEGIN
[txxxxx] (text from child task)
[txxxxx] END
```

The output from each task includes one BEGIN line and one END line with whatever the task prints in between.

In C, the output file descriptor may be specified. Giving a null pointer turns output collection off. [Note file option not implemented in PVM 3.3.0 output goes to calling task's stdout]

In Fortran, output collection can only be turned on or off, and is logged to stdout of the parent task.

If `pvm_exit` is called 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, output collection can be turned off by calling pvm_catchout(0) before calling pvm_exit.

pvm_catchout() always returns PvmOk.

Examples

C:

```
#include <stdio.h>
pvm_catchout(stdout);
```

Fortran:

```
CALL PVMFCATCHOUT( 1 )
```

Errors

No error conditions are returned by pvm_catchout

pvmfconfig() pvm_config()

Returns information about the present virtual machine configuration.

Synopsis

C

```
int info = pvm_config( int *nhost, int *narch,
            struct pvmhostinfo **hostp )
struct pvmhostinfo{
    int  hi_tid;
    char *hi_name;
    char *hi_arch;
    int  hi_speed;
} hostp;
```

Fortran

```
call pvmfconfig( nhost, narch, dtid,
name, arch, speed, info )
```

Parameters

- `nhost` – integer returning the number of hosts (pvmds) in the virtual machine.
- `narch` – integer returning the number of different data formats being used.
- `hostp` – pointer to an array of structures that contain information about each host, including its pvmd task ID, name, architecture, and relative speed.
- `dtid` – Integer returning pvmd task ID for this host.
- `name` – Character string returning name of this host.
- `arch` – Character string returning name of host architecture.
- `speed` – Integer returning relative speed of this host. Default value is 1000.
- `info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_config` returns information about the present virtual machine. The information returned is similar to that available from the console command `conf`. The C function returns information about the entire virtual machine in one call. The Fortran function returns information about one host per call and cycles through all the hosts. Thus, if pvmfconfig is called nhost times, the entire virtual machine will be represented. If pvm_config is successful, `info` will be 0. If some error occurs, `info` will be < 0.

Examples

C:

```
info = pvm_config( &nhost, &narch, &hostp );
```

Fortran:

```
Do i=1, NHOST
  CALL PVMFCONFIG( NHOST,NARCH,DTID(i),HOST(i),ARCH(i),
                   SPEED(i),INFO )
Enddo
```

Errors

The following error condition can be returned by `pvm_config`

Name	Possible Cause
PvmSysErr	pvmd not responding.

pvmfdelhost() pvm_delhosts()

deletes one or more hosts from the virtual machine.

Synopsis

C	`int info = pvm_delhosts( char **hosts, int nhost, int *infos )`
Fortran	`call pvmfdelhost( host, info )`

Parameters

`hosts`	–	an array of pointers to character strings containing the names of the machines to be deleted.
`nhost`	–	integer specifying the number of hosts to be deleted.
`infos`	–	integer array of length `nhost` which contains the status code returned by the routine for the individual hosts. Values less than zero indicate an error.
`host`	–	character string containing the name of the machine to be deleted.
`info`	–	integer status code returned by the routine. Values less than nhost indicate partial failure, values less than 1 indicate total failure.

Discussion

The routine `pvm_delhosts` deletes the computers pointed to in `hosts`, from the existing configuration of computers making up the virtual machine. All PVM processes and the pvmd running on these computers are killed as the computer is deleted. If pvm_delhosts is successful, `info` will be `nhost`. Partial success is indicated by 1 <= `info` < `nhost`, and total failure by `info` < 1. The array `infos` can be checked to determine which host caused the error.

The Fortran routine `pvmfdelhost` deletes a single host from the configuration with each call.

If a host fails, the PVM system will continue to function and will automatically delete this host from the virtual machine. An application can be notified of a host failure by calling pvm_notify. It is still the responsibility of the application developer to make his application tolerant of host failure.

Examples

C:

```
static char *hosts[] = {
            "sparky",
            "thud.cs.utk.edu",
};
info = pvm_delhosts( hosts, 2 );
```

Fortran:

```
CALL PVMFDELHOST( 'azure', INFO )
```

Errors

These error conditions can be returned by `pvm_delhosts`

Name	Possible cause
PvmBadParam	giving an invalid argument value.
PvmSysErr	local pvmd not responding.
PvmOutOfRes	PVM has run out of system resources.

pvmfexit() pvm_exit()

tells the local pvmd that this process is leaving PVM.

Synopsis

C	`int info = pvm_exit( void )`
Fortran	`call pvmfexit( info )`

Parameters

`info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

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 serial process.

Pvm_exit should be called by all PVM processes before they stop or exit for good. It *must* be called by processes that were not started with pvm_spawn.

Examples

C:

```
/* Program done */
pvm_exit();
exit();
```

Fortran:

```
CALL PVMFEXIT(INFO)
STOP
```

Errors

Name	Possible cause
PvmSysErr	pvmd not responding

pvmffreebuf() pvm_freebuf()

disposes of a message buffer.

Synopsis

C	`int info = pvm_freebuf( int bufid )`
Fortran	`call pvmffreebuf( bufid, info )`

Parameters

`bufid` – integer message buffer identifier.

`info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_freebuf` frees the memory associated with the message buffer identified by `bufid`. Message buffers are created by pvm_mkbuf, pvm_initsend, and pvm_recv. If pvm_freebuf is successful, `info` will be 0. If some error occurs then `info` will be < 0.

pvm_freebuf can be called for a send buffer created by pvm_mkbuf after the message has been sent and is no longer needed.

Receive buffers typically do not have to be freed unless they have been saved in the course of using multiple buffers. But pvm_freebuf can be used to destroy receive buffers as well. So messages that have arrived but are no longer needed due to some other event in an application can be destroyed so they will not consume buffer space.

Typically multiple send and receive buffers are not needed and the user can simply use the pvm_initsend routine to reset the default send buffer.

There are several cases where multiple buffers are useful. One example where multiple message buffers are needed involves libraries or graphical interfaces that use PVM and interact with a running PVM application but do not want to interfere with the application's own communication.

When multiple buffers are used they generally are made and freed for each message that is packed. In fact, pvm_initsend simply does a pvm_freebuf followed by a pvm_mkbuf for the default buffer.

Examples

C:

```
bufid = pvm_mkbuf( PvmDataDefault );
      :
info = pvm_freebuf( bufid );
```

Fortran:

```
CALL PVMFMKBUF( PVMDEFAULT, BUFID )
      :
CALL PVMFFREEBUF( BUFID, INFO )
```

Errors

These error conditions can be returned by `pvm_freebuf`

Name	Possible cause
PvmBadParam	giving an invalid argument value.
PvmNoSuchBuf	giving an invalid bufid value.

pvmfgather() pvm_gather()

A specified member of the group gathers data from each member of the group into a single array.

Synopsis

C
```
int info = pvm_gather( void *result,  void *data,
                 int count, int datatype, int msgtag,
                 char *group, int rootginst)
```

Fortran
```
call pvmfgather(result, data, count, datatype,
msgtag, group, rootginst, info)
```

Parameters

`result`	–	On the root this is a pointer to the starting address of an array `datatype` of local values which are to be accumulated from the members of the group. This array should be of length at least equal to the number of group members. times `count`. This argument is significant only on the root.
`data`	–	For each group member this is a pointer to the starting address of an array of length `count` which will be sent to the specified root member of the group.
`count`	–	Integer specifying the number of elements of type `datatype` to be sent by each member of the group to the root.
`datatype`	–	Integer specifying the type of the entries in the result and data arrays. For a list of supported types see pvm_psend().
`msgtag`	–	Integer message tag supplied by the user. msgtag should be >= 0.
`group`	–	Character string group name of an existing group.
`rootginst`	–	Integer instance number of group member who performs the gather of the messages from the members of the group.
`info`	–	Integer status code returned by the routine. Values less than zero indicate an error.

Discussion

pvm_gather() gathers data from each member of the group to a single member of the group, specified by rootginst. All group members must call pvm_gather(), each sends its array of length count of datatype to the root which concatenates these messages in order relative to the sender's instance number in the group. Thus the first count entries in the result array will be the data from group member 1, the next count entries from group member 2, and so on.

In using the scatter and gather routines, keep in mind that C stores multidimensional arrays in row order, typically starting with an initial index of 0; whereas, Fortran stores arrays in column order, typically starting with an index of 1.

Note: pvm_gather() does not block. If a task calls pvm_gather and then leaves the group before the root has called pvm_gather an error may occur.

The current algorithm is very simple and robust. Future implementations will make more efficient use of the architecture to allow greater parallelism.

Examples

C:

```
info =  pvm_gather(&getmatrix, &myrow, 10, PVM_INT,
             msgtag, "workers", rootginst);
```

Fortran:

```
CALL PVMFGATHER(GETMATRIX, MYCOLUMN, COUNT, INTEGER4,
        MTAG, 'workers', ROOT, INFO)
```

Errors

These error conditions can be returned by `pvm_gather`

Name	Possible cause
PvmBadParam	giving an invalid argument value.
PvmNoInst	Calling task is not in the group.
PvmSysErr	local pvmd is not responding.

pvmfgetinst() pvm_getinst()

returns the instance number in a group of a PVM process.

Synopsis

C	`int inum = pvm_getinst( char *group, int tid )`
Fortran	`call pvmfgetinst( group, tid, inum )`

Parameters

- `group` – character string group name of an existing group.
- `tid` – integer task identifier of a PVM process.
- `inum` – integer instance number returned by the routine. Instance numbers start at 0 and count up. Values less than zero indicate an error.

Discussion

The routine `pvm_getinst` takes a group name `group` and a PVM task identifier `tid` and returns the unique instance number that corresponds to the input. If pvm_getinst is successful, `inum` will be >= 0. If some error occurs then `inum` will be < 0.

Examples

C:

```
inum = pvm_getinst( "worker", pvm_mytid() );
--------
inum = pvm_getinst( "worker", tid[i] );
```

Fortran:

```
CALL PVMFGETINST( 'GROUP3', TID, INUM )
```

Errors

These error conditions can be returned by `pvm_getinst`

Name	Possible cause
PvmSysErr	pvmd was not started or has crashed.
PvmBadParam	giving an invalid tid value.
PvmNoGroup	giving a non-existent group name.
PvmNotInGroup	specifying a group in which the tid is not a member.

pvmfgetopt() pvm_getopt()

Shows various libpvm options

Synopsis

C	`int val = pvm_getopt( int what )`
Fortran	`call pvmfgetrbuf( what, val )`

Parameters

`what` – Integer defining what to get. Options include:

Option	value	MEANING
`PvmRoute`	1	routing policy
`PvmDebugMask`	2	debugmask
`PvmAutoErr`	3	auto error reporting
`PvmOutputTid`	4	stdout device for children
`PvmOutputCode`	5	output msgtag
`PvmTraceTid`	6	trace device for children
`PvmTraceCode`	7	trace msgtag
`PvmFragSize`	8	message fragment size
`PvmResvTids`	9	Allow use of reserved msgtags and TIDs

`val` – Integer specifying value of option. Predefined route values are:

Option	value	MEANING
`PvmDontRoute`	1	
`PvmAllowDirect`	2	
`PvmRouteDirect`	3	

Discussion

The routine `pvm_getopt` allows the user to see the value of options set in PVM. See pvm_setopt for a description of options that can be set.

Examples

C:

```
route_method = pvm_getopt( PvmRoute );
```

Fortran:

```
CALL PVMFGETOPT( PVMAUTOERR, VAL )
```

Errors

These error conditions can be returned by `pvm_getopt`

Name	Possible cause
PvmBadParam	giving an invalid argument.

pvmfgetrbuf() pvm_getrbuf()

returns the message buffer identifier for the active receive buffer.

Synopsis

C `int bufid = pvm_getrbuf( void )`

Fortran `call pvmfgetrbuf( bufid )`

Parameters

`bufid` – integer the returned message buffer identifier for the active receive buffer.

Discussion

The routine `pvm_getrbuf` returns the message buffer identifier `bufid` for the active receive buffer or 0 if there is no current buffer.

Examples

C:

```
bufid = pvm_getrbuf();
```

Fortran:

```
CALL PVMFGETRBUF( BUFID )
```

Errors

No error conditions are returned by `pvm_getrbuf`

pvmfgetsbuf() pvm_getsbuf()

returns the message buffer identifier for the active send buffer.

Synopsis

C	`int bufid = pvm_getsbuf( void )`
Fortran	`call pvmfgetsbuf( bufid )`

Parameters

`bufid` – integer the returned message buffer identifier for the active send buffer.

Discussion

The routine `pvm_getsbuf` returns the message buffer identifier `bufid` for the active send buffer or 0 if there is no current buffer.

Examples

C:

```
bufid = pvm_getsbuf();
```

Fortran:

```
CALL PVMFGETSBUF( BUFID )
```

Errors

No error conditions are returned by `pvm_getsbuf`

pvmfgettid() pvm_gettid()

returns the tid of the process identified by a group name and instance number.

Synopsis

C	`int tid = pvm_gettid( char *group, int inum )`
Fortran	`call pvmfgettid( group, inum, tid )`

Parameters

`group`	–	character string that contains the name of an existing group.
`inum`	–	integer instance number of the process in the group.
`tid`	–	integer task identifier returned.

Discussion

The routine `pvm_gettid` returns the tid of the PVM process identified by the group name `group` and the instance number `inum`. If pvm_gettid is successful, `tid` will be > 0. If some error occurs then `tid` will be < 0.

Examples

C:

```
tid = pvm_gettid("worker",0);
```

Fortran:

```
 CALL PVMFGETTID('worker',5,TID)
```

Errors

These error conditions can be returned by `pvm_gettid`.

Name	Possible cause
PvmSysErr	Can not contact the local pvmd most likely it is not running.
PvmBadParam	Bad Parameter most likely a NULL character string.
PvmNoGroup	No group exists by that name.
PvmNoInst	No such instance in the group.

pvmfgsize() pvm_gsize()

returns the number of members presently in the named group.

Synopsis

C	`int size = pvm_gsize( char *group )`
Fortran	`call pvmfgsize( group, size )`

Parameters

`group`	–	character string group name of an existing group.
`size`	–	integer returning the number of members presently in the group. Values less than zero indicate an error.

Discussion

The routine `pvm_gsize` returns the size of the group named `group`. If there is an error `size` will be negative.

Since groups can change dynamically in PVM 3, this routine can only guarantee to return the instantaneous size of a given group.

Examples

C:

```
size = pvm_gsize( "worker" );
```

Fortran:

```
CALL PVMFGSIZE( 'group2', SIZE )
```

Errors

These error conditions can be returned by `pvm_gsize`

Name	Possible cause
PvmSysErr	pvmd was not started or has crashed.
PvmBadParam	giving an invalid group name.

pvmfhalt pvm_halt()

Shuts down the entire PVM system.

Synopsis

C	`int info = pvm_halt( void )`
Fortran	`call pvmfhalt( info )`

Parameters

`info` – Integer returns the error status.

Discussion

The routine `pvm_halt` shuts down the entire PVM system including remote tasks, remote pvmd, the local tasks (including the calling task) and the local pvmd.

Errors

The following error condition can be returned by `pvm_halt`

Name	Possible cause
PvmSysErr	local pvmd is not responding.

pvmfhostsync() pvm_hostsync()

Get time-of-day clock from PVM host.

Synopsis

C
```
#include <sys/time.h>
int info = pvm_hostsync( int host, struct timeval *clk,
struct timeval *delta )
```
Fortran
```
call pvmfhostsync( host, clksec, clkusec,
deltasec, deltausec, info )
```

Parameters

`host`	–	TID of host.
`clk` or `clksec` and `clkusec`)	–	Returns time-of-day clock sample from host.
`delta` or `deltasec` and `deltausec`)	–	Returns difference between local clock and remote host clock.

Discussion

`pvm_hostsync()` samples the time-of day clock of a host in the virtual machine and returns both the clock value and the difference between local and remote clocks.

To reduce the delta error due to message transit time, local clock samples are taken before and after reading the remote clock. Delta is the difference between the mean local clocks and remote clock.

Note that the delta time can be negative. The microseconds field is always normalized to 0..999999, while the sign of the seconds field gives the sign of the delta.

In C, if `clk` or `delta` is input as a null pointer, that parameter is not returned.

Errors

The following error conditions can be returned by `pvm_synchost`

Name	Possible cause
PvmSysErr	local pvmd is not responding.
PvmNoHost	no such host.
PvmHostFail	host is unreachable (and thus possibly failed).

pvmfinitsend() pvm_initsend()

clear default send buffer and specify message encoding.

Synopsis

C	`int bufid = pvm_initsend( int encoding )`
Fortran	`call pvmfinitsend( encoding, bufid )`

Parameters

`encoding` – integer specifying the next message's encoding scheme.
Options in C are:

Encoding value		MEANING
`PvmDataDefault`	0	XDR
`PvmDataRaw`	1	no encoding
`PvmDataInPlace`	2	data left in place

`bufid` – integer returned containing the message buffer identifier. Values less than zero indicate an error.

Discussion

The routine `pvm_initsend` clears the send buffer and prepares it for packing a new message. The encoding scheme used for this packing is set by `encoding`. XDR encoding is used by default because PVM can not know if the user is going to add a heterogeneous machine before this message is sent. If the user knows that the next message will only be sent to a machine that understands the native format, then he can use *PvmDataRaw* encoding and save on encoding costs.

PvmDataInPlace encoding specifies that data be left in place during packing. The message buffer only contains the sizes and pointers to the items to be sent. When pvm_send is called the items are copied directly out of the user's memory. This option decreases the number of times a message is copied at the expense of requiring the user to not modify the items between the time they are packed and the time they are sent. The PvmDataInPlace is not implemented in PVM 3.2.

If pvm_initsend is successful, then `bufid` will contain the message buffer identifier. If some error occurs then `bufid` will be < 0.

See also pvm_mkbuf.

Examples

C:

```
bufid = pvm_initsend( PvmDataDefault );
info = pvm_pkint( array, 10, 1 );
msgtag = 3 ;
info = pvm_send( tid, msgtag );
```

Fortran:

```
CALL PVMFINITSEND(PVMRAW, BUFID)
CALL PVMFPACK( REAL4, DATA, 100, 1, INFO )
CALL PVMFSEND( TID, 3, INFO )
```

Errors

These error conditions can be returned by `pvm_initsend`

Name	Possible cause
PvmBadParam	giving an invalid encoding value
PvmNoMem	Malloc has failed. There is not enough memory to create the buffer

pvmfjoingroup() pvm_joingroup()

enrolls the calling process in a named group.

Synopsis

C	`int inum = pvm_joingroup( char *group )`
Fortran	`call pvmfjoingroup( group, inum )`

Parameters

- `group` – character string group name of an existing group.
- `inum` – integer instance number returned by the routine. Instance numbers start at 0 and count up. Values less than zero indicate an error.

Discussion

The routine `pvm_joingroup` enrolls the calling task in the group named `group` and returns the instance number `inum` of this task in this group. If there is an error `inum` will be negative.

Instance numbers start at 0 and count up. When using groups a (group, inum) pair uniquely identifies a PVM process. This is consistent with the previous PVM naming schemes. If a task leaves a group by calling pvm_lvgroup and later rejoins the same group, the task is not guaranteed to get the same instance number. PVM attempts to reuse old instance numbers, so when a task joins a group it will get the lowest available instance number. A PVM 3 task can be a member of multiple groups simultaneously.

Examples

C:

```
inum = pvm_joingroup( "worker" );
```

Fortran:

```
CALL PVMFJOINGROUP( 'group2', INUM )
```

Errors

These error conditions can be returned by `pvm_joingroup`

Name	Possible cause
PvmSysErr	pvmd was not started or has crashed.
PvmBadParam	giving a NULL group name.
PvmDupGroup	trying to join a group you are already in.

pvmfkill() pvm_kill()

terminates a specified PVM process.

Synopsis

C	`int info = pvm_kill( int tid )`
Fortran	`call pvmfkill( tid, info )`

Parameters

- `tid` – integer task identifier of the PVM process to be killed (not yourself).
- `info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_kill` sends a terminate (SIGTERM) signal to the PVM process identified by `tid`. In the case of multiprocessors the terminate signal is replaced with a host dependent method for killing a process. If pvm_kill is successful, `info` will be 0. If some error occurs then `info` will be < 0.

pvm_kill is not designed to kill the calling process. To kill yourself in C call pvm_exit() followed by exit(). To kill yourself in Fortran call pvmfexit followed by stop.

Examples

C:

```
info = pvm_kill( tid );
```

Fortran:

```
CALL PVMFKILL( TID, INFO )
```

Errors

These error conditions can be returned by `pvm_kill`

Name	Possible cause
PvmBadParam	giving an invalid tid value.
PvmSysErr	pvmd not responding.

pvmflvgroup() pvm_lvgroup()

unenrolls the calling process from a named group.

Synopsis

C	`int info = pvm_lvgroup( char *group )`
Fortran	`call pvmflvgroup( group, info )`

Parameters

- `group` – character string group name of an existing group.
- `info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_lvgroup` unenrolls the calling process from the group named `group`. If there is an error `info` will be negative.

If a process leaves a group by calling either pvm_lvgroup or pvm_exit, and later rejoins the same group, the process may be assigned a new instance number. Old instance numbers are reassigned to processes calling pvm_joingroup.

Examples

C:

```
info = pvm_lvgroup( "worker" );
```

Fortran:

```
CALL PVMFLVGROUP( 'group2', INFO )
```

Errors

These error conditions can be returned by `pvm_lvgroup`

Name	Possible cause
PvmSysErr	pvmd not responding.
PvmBadParam	giving a NULL group name.
PvmNoGroup	giving a non-existent group name.
PvmNotInGroup	asking to leave a group you are not a member of.

pvmfmcast() pvm_mcast()

multicasts the data in the active message buffer to a set of tasks.

Synopsis

C	`int info = pvm_mcast( int *tids, int ntask, int msgtag )`
Fortran	`call pvmfmcast( ntask, tids, msgtag, info )`

Parameters

`ntask`	–	integer specifying the number of tasks to be sent to.
`tids`	–	integer array of length at least `ntask` containing the task IDs of the tasks to be sent to.
`msgtag`	–	integer message tag supplied by the user. msgtag should be >= 0.
`info`	–	integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_mcast` multicasts a message stored in the active send buffer to `ntask` tasks specified in the `tids` array. The message is not sent to the caller even if its tid is in `tids`.. The content of the message can be distinguished by `msgtag`. If pvm_mcast is successful, `info` will be 0. If some error occurs then `info` will be < 0.

The receiving processes can call either pvm_recv or pvm_nrecv to receive their copy of the multicast. pvm_mcast is asynchronous. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processors. This is in contrast to synchronous communication, during which computation on the sending processor halts until the matching receive is executed by the receiving processor.

pvm_mcast first determines which other pvmds contain the specified tasks. Then passes the message to these pvmds which in turn distribute the message to their local tasks without further network traffic.

Multicasting is not supported by most multiprocessor vendors. Typically their native calls only support broadcasting to *all* the user's processes on a multiprocessor. Because of this omission, pvm_mcast may not be an efficient communication method on some multiprocessors except in the special case of broadcasting to all PVM processes.

Examples

C:

```
info = pvm_initsend( PvmDataRaw );
info = pvm_pkint( array, 10, 1 );
msgtag = 5 ;
info = pvm_mcast( tids, ntask, msgtag );
```

Fortran:

```
CALL PVMFINITSEND(PVMDEFAULT)
CALL PVMFPACK( REAL4, DATA, 100, 1, INFO )
CALL PVMFMCAST( NPROC, TIDS, 5, INFO )
```

Errors

These error conditions can be returned by `pvm_mcast`

Name	Possible cause
PvmBadParam	giving a msgtag < 0.
PvmSysErr	pvmd not responding.
PvmNoBuf	no send buffer.

pvmfmkbuf() pvm_mkbuf()

creates a new message buffer.

Synopsis

C	`int bufid = pvm_mkbuf( int encoding )`
Fortran	`call pvmfmkbuf( encoding, bufid )`

Parameters

`encoding` – integer specifying the buffer's encoding scheme.
Options in C are:

Encoding value		MEANING
`PvmDataDefault`	0	XDR
`PvmDataRaw`	1	no encoding
`PvmDataInPlace`	2	data left in place

`bufid` – integer message buffer identifier returned. Values less than zero indicate an error.

Discussion

The routine `pvm_mkbuf` creates a new message buffer and sets its encoding status to `encoding`. If pvm_mkbuf is successful, `bufid` will be the identifier for the new buffer, which can be used as a send buffer. If some error occurs then `bufid` will be < 0.

With the default setting XDR encoding is used when packing the message because PVM can not know if the user is going to add a heterogeneous machine before this message is sent. The other options to encoding allow the user to take advantage of knowledge about his virtual machine even when it is heterogeneous. For example, if the user knows that the next message will only be sent to a machine that understands the native format, then he can use `PvmDataRaw` encoding and save on encoding costs.

PvmDataInPlace encoding specifies that data be left in place during packing. The message buffer only contains the sizes and pointers to the items to be sent. When pvm_send is called the items are copied directly out of the user's memory. This option decreases the number of times a message is copied at the expense of requiring the user to not modify the items between the time they are packed and the time they are sent. The PvmDataInPlace is also not implemented in PVM 3.2.

pvm_mkbuf is required if the user wishes to manage multiple message buffers and should be used in conjunction with pvm_freebuf. pvm_freebuf should be called for a send buffer after a message has been sent and is no longer needed.

Receive buffers are created automatically by the pvm_recv and pvm_nrecv routines and do not have to be freed unless they have been explicitly saved with pvm_setrbuf.

Typically multiple send and receive buffers are not needed and the user can simply use the pvm_initsend routine to reset the default send buffer.

There are several cases where multiple buffers are useful. One example where multiple message buffers are needed involves libraries or graphical interfaces that use PVM and interact with a running PVM application but do not want to interfere with the application's own communication.

When multiple buffers are used they generally are made and freed for each message that is packed.

Examples

C:

```
bufid = pvm_mkbuf( PvmDataRaw );
  /* send message */
info = pvm_freebuf( bufid );
```

Fortran:

```
          CALL PVMFMKBUF(PVMDEFAULT, MBUF)
*          SEND MESSAGE HERE
          CALL PVMFFREEBUF( MBUF, INFO )
```

Errors

These error condition can be returned by `pvm_mkbuf`

Name	Possible cause
PvmBadParam	giving an invalid encoding value.
PvmNoMem	Malloc has failed. There is not enough memory to create the buffer

pvmfmstat() pvm_mstat()

returns the status of a host in the virtual machine.

Synopsis

C	`int mstat = pvm_mstat( char *host )`
Fortran	`call pvmfmstat( host, mstat )`

Parameters

`host` – character string containing the host name.

`mstat` – integer returning machine status:

value	MEANING
PvmOk	host is OK
PvmNoHost	host is not in virtual machine
PvmHostFail	host is unreachable (and thus possibly failed)

Discussion

The routine `pvm_mstat` returns the status `mstat` of the computer named `host` with respect to running PVM processes. This routine can be used to determine if a particular host has failed and if the virtual machine needs to be reconfigured.

Examples

C:

```
mstat = pvm_mstat( "msr.ornl.gov" );
```

Fortran:

```
CALL PVMFMSTAT( 'msr.ornl.gov', MSTAT )
```

Errors

These error conditions can be returned by `pvm_mstat`

Name	Possible cause
PvmSysErr	pvmd not responding.
PvmNoHost	giving a host name not in the virtual machine.
PvmHostFail	host is unreachable (and thus possibly failed).

pvmfmytid() pvm_mytid()

returns the *tid* of the process

Synopsis

C	`int tid = pvm_mytid( void )`
Fortran	`call pvmfmytid( tid )`

Parameters

`tid` – integer task identifier of the calling PVM process is returned. Values less than zero indicate an error.

Discussion

The routine enrolls this process into PVM on its first call and generates a unique `tid` if this process was not created by pvm_spawn. pvm_mytid returns the `tid` of the calling process and can be called multiple times in an application. Any PVM system call (not just pvm_mytid) will enroll a task in PVM if the task is not enrolled before the call.

The tid is a 32 bit positive integer created by the local pvmd. The 32 bits are divided into fields that encode various information about this process such as its location in the virtual machine (i.e. local pvmd address), the CPU number in the case where the process is on a multiprocessor, and a process ID field. This information is used by PVM and is not expected to be used by applications.

If PVM has not been started before an application calls pvm_mytid the returned `tid` will be < 0.

Examples

C:

```
tid = pvm_mytid( );
```

Fortran:

```
CALL PVMFMYTID( TID )
```

Errors

This error condition can be returned by **pvm_mytid**

Name	Possible cause
PvmSysErr	pvmd not responding.

pvmfnotify() pvm_notify()

Request notification of PVM event such as host failure.

Synopsis

C
```
int info = pvm_notify( int what, int msgtag,
                       int cnt, int *tids )
```
Fortran
```
call pvmfnotify( what, msgtag, cnt, tids, info )
```

Parameters

`what` – integer identifier of what event should trigger the notification. Presently the options are:

value	MEANING
PvmTaskExit	notify if task exits
PvmHostDelete	notify if host is deleted
PvmHostAdd	notify if host is added

`msgtag` – integer message tag to be used in notification.

`cnt` – integer specifying the length of the tids array for PvmTaskExit and PvmHostDelete. For PvmHostAdd specifies the number of times to notify.

`tids` – integer array of length `ntask` that contains a list of task or pvmd tids to be notified. The array should be empty with the PvmHostAdd option.

`info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_notify` requests PVM to notify the caller on detecting certain events. In response to a notify request, some number of messages (see below) are sent by PVM back to the calling task. The messages are tagged with the code (msgtag) supplied to notify.

The tids array specifies who to monitor when using TaskExit or HostDelete, it contains nothing when using HostAdd. If required, the routines pvm_config and pvm_tasks can be used to obtain task and pvmd tids.

The notification messages have the following format:

`PvmTaskExit` One notify message for each tid requested. The message body contains a single tid of exited task.

`PvmHostDelete` One message for each tid requested. The message body contains a single pvmd-tid of exited pvmd.

`PvmHostAdd` Up to `cnt` notify messages are sent. The message body contains an integer count followed by a list of pvmd-tids of the new pvmds. The counter of

PvmHostAdd messages remaining is updated by successive calls to pvm_notify. Specifying a cnt of -1 turns on PvmHostAdd messages until a future notify; a count of zero disables them.

Tids in the notify messages are packed as integers.

The calling task(s) are responsible for receiving the message with the specified msgtag and taking appropriate action. Future versions of PVM may expand the list of available notification events.

Note that the notify request is "consumed" - e.g. a PvmHostAdd request generates a single reply message.

Examples

C:

```
info = pvm_notify( PvmHostAdd, 9999, 1, dummy )
```

Fortran:

```
CALL PVMFNOTIFY( PVMHOSTDELETE, 1111, NPROC, TIDS, INFO )
```

Errors

Name	Possible cause
PvmSysErr	pvmd not responding.
PvmBadParam	giving an invalid argument value.

pvmfnrecv() pvm_nrecv()

non-blocking receive.

Synopsis

C	`int bufid = pvm_nrecv( int tid, int msgtag )`
Fortran	`call pvmfnrecv( tid, msgtag, bufid )`

Parameters

- `tid` – integer task identifier of sending process supplied by the user. (A -1 in this argument matches any tid (wildcard).)
- `msgtag` – integer message tag supplied by the user. msgtag should be >= 0. (A -1 in this argument matches any message tag (wildcard).)
- `bufid` – integer returning the value of the new active receive buffer identifier. Values less than zero indicate an error.

Discussion

The routine `pvm_nrecv` checks to see if a message with label `msgtag` has arrived from `tid`. If a matching message has arrived pvm_nrecv immediately places the message in a new *active* receive buffer, which also clears the current receive buffer if any, and returns the buffer identifier in `bufid`.

If the requested message has not arrived, then pvm_nrecv immediately returns with a 0 in `bufid`. If some error occurs `bufid` will be < 0.

A -1 in `msgtag` or `tid` matches anything. This allows the user the following options. If tid $= -1$ and msgtag is defined by the user, then pvm_nrecv will accept a message from any process which has a matching msgtag. If msgtag $= -1$ and tid is defined by the user, then pvm_nrecv will accept any message that is sent from process tid. If tid $= -1$ and msgtag $= -1$, then pvm_nrecv will accept any message from any process.

pvm_nrecv is non-blocking in the sense that the routine always returns immediately either with the message or with the information that the message has not arrived at the local pvmd yet. pvm_nrecv can be called multiple times to check if a given message has arrived yet. In addition pvm_recv can be called for the same message if the application runs out of work it could do before receiving the data.

If pvm_nrecv returns with the message, then the data in the message can be unpacked into the user's memory using the unpack routines.

The PVM model guarantees the following about message order. 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.

Examples

C:

```
tid = pvm_parent();
msgtag = 4 ;
arrived = pvm_nrecv( tid, msgtag );
if( arrived > 0 )
  info = pvm_upkint( tid_array, 10, 1 );
else
/* go do other computing */
```

Fortran:

```
          CALL PVMFNRECV( -1, 4, ARRIVED )
          IF ( ARRIVED .GT. 0 ) THEN
             CALL PVMFUNPACK( INTEGER4, TIDS, 25, 1, INFO )
             CALL PVMFUNPACK( REAL8, MATRIX, 100, 100, INFO )
          ELSE
*            GO DO USEFUL WORK
          ENDIF
```

Errors

These error conditions can be returned by `pvm_nrecv`.

Name	Possible cause
PvmBadParam	giving an invalid tid value or msgtag.
PvmSysErr	pvmd not responding.

pvmfpack() pvm_pk*()

pack the active message buffer with arrays of prescribed data type.

Synopsis

C

```
int info = pvm_packf( const char *fmt, ... )
int info = pvm_pkbyte(  char   *xp, int nitem, int stride )
int info = pvm_pkcplx(  float  *cp, int nitem, int stride )
int info = pvm_pkdcplx( double *zp, int nitem, int stride )
int info = pvm_pkdouble(double *dp, int nitem, int stride )
int info = pvm_pkfloat( float  *fp, int nitem, int stride )
int info = pvm_pkint(   int    *ip, int nitem, int stride )
int info = pvm_pkuint( unsigned int *ip, int nitem, int stride )
int info = pvm_pkushort( unsigned short *ip, int nitem, int stride )
int info = pvm_pkulong( unsigned long *ip, int nitem, int stride )
int info = pvm_pklong(  long   *ip, int nitem, int stride )
int info = pvm_pkshort( short  *jp, int nitem, int stride )
int info = pvm_pkstr(   char   *sp )
```

Fortran

```
call pvmfpack( what, xp, nitem, stride, info )
```

Parameters

- `fmt` – Printflike format expression specifying what to pack. (See discussion).
- `nitem` – The total number of *items* to be packed (not the number of bytes).
- `stride` – The stride to be used when packing the items. For example, if stride= 2 in pvm_pkcplx, then every other complex number will be packed.
- `xp` – pointer to the beginning of a block of bytes. Can be any data type, but must match the corresponding unpack data type.
- `cp` – complex array at least nitem*stride items long.
- `zp` – double precision complex array at least nitem*stride items long.
- `dp` – double precision real array at least nitem*stride items long.
- `fp` – real array at least nitem*stride items long.
- `ip` – integer array at least nitem*stride items long.
- `jp` – integer*2 array at least nitem*stride items long.
- `sp` – pointer to a null terminated character string.

`what` – integer specifying the type of data being packed.

what options			
STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

`info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

Each of the `pvm_pk*` routines packs an array of the given data type into the active send buffer. The arguments for each of the routines are a pointer to the first item to be packed, `nitem` which is the total number of items to pack from this array, and `stride` which is the stride to use when packing.

An exception is pvm_pkstr() which by definition packs a NULL terminated character string and thus does not need `nitem` or `stride` arguments. The Fortran routine pvmfpack(STRING, ...) expects `nitem` to be the number of characters in the string and `stride` to be 1.

If the packing is successful, `info` will be 0. If some error occurs then `info` will be < 0.

A single variable (not an array) can be packed by setting `nitem`$= 1$ and `stride`$=$ 1. C structures have to be packed one data type at a time.

The routine pvm_packf() uses a printflike format expression to specify what and how to pack data into the send buffer. All variables are passed as addresses if count and stride are specified; otherwise, variables are assumed to be values. A BNF-like description of the format syntax is:

```
format : null | init | format fmt
init : null | '%' '+'
fmt : '%' count stride modifiers fchar
fchar : 'c' | 'd' | 'f' | 'x' | 's'
count : null | [0-9]+ | '*'
stride : null | '.' ( [0-9]+ | '*' )
modifiers : null | modifiers mchar
mchar : 'h' | 'l' | 'u'
```

```
Formats:
  +  means initsend - must match an int (how) in the param list.
  c  pack/unpack bytes
  d  integers
  f  float
  x  complex float
  s  string
```

```
Modifiers:
  h  short (int)
  l  long  (int, float, complex float)
  u  unsigned (int)

'*' count or stride must match an int in the param list.
```

Future extensions to the `what` argument in pvmfpack will include 64 bit types when XDR encoding of these types is available. Meanwhile users should be aware that precision can be lost when passing data from a 64 bit machine like a Cray to a 32 bit machine like a SPARCstation. As a mnemonic the `what` argument name includes the number of bytes of precision to expect. By setting encoding to PVMRAW (see pvmfinitsend) data can be transferred between two 64 bit machines with full precision even if the PVM configuration is heterogeneous.

Messages should be unpacked exactly like they were packed to insure data integrity. Packing integers and unpacking them as floats will often fail because a type encoding will have occurred transferring the data between heterogeneous hosts. Packing 10 integers and 100 floats then trying to unpack only 3 integers and the 100 floats will also fail.

Examples

C:

```
info = pvm_initsend( PvmDataDefault );
info = pvm_pkstr( "initial data" );
info = pvm_pkint( &size, 1, 1 );
info = pvm_pkint( array, size, 1 );
info = pvm_pkdouble( matrix, size*size, 1 );
msgtag = 3 ;
info = pvm_send( tid, msgtag );
```

Fortran:

```
CALL PVMFINITSEND(PVMRAW, INFO)
CALL PVMFPACK( INTEGER4, NSIZE, 1, 1, INFO )
CALL PVMFPACK( STRING, 'row 5 of NXN matrix', 19, 1, INFO )
CALL PVMFPACK( REAL8, A(5,1), NSIZE, NSIZE , INFO )
CALL PVMFSEND( TID, MSGTAG, INFO )
```

Errors

Name	Possible cause
PvmNoMem	Malloc has failed. Message buffer size has exceeded the available memory on this host.
PvmNoBuf	There is no active send buffer to pack into. Try calling pvm_initsend before packing message.

pvmfparent() pvm_parent()

returns the tid of the process that spawned the calling process.

Synopsis

C `int tid = pvm_parent( void )`
Fortran `call pvmfparent( tid )`

Parameters

`tid` – integer returns the task identifier of the parent of the calling process. If the calling process was not created with pvm_spawn, then tid = PvmNoParent.

Discussion

The routine `pvm_parent` returns the `tid` of the process that spawned the calling process. If the calling process was not created with pvm_spawn, then `tid` is set to PvmNoParent.

Examples

C:

```
tid = pvm_parent();
```

Fortran:

```
CALL PVMFPARENT( TID )
```

Errors

this error condition can be returned by `pvm_parent`.

Name	Possible cause
PvmNoParent	The calling process was not created with pvm_spawn.

pvmfperror() pvm_perror()

prints the error status of the last PVM call.

Synopsis

C	`int info = pvm_perror( char *msg )`
Fortran	`call pvmfperror( msg, info )`

Parameters

- `msg` – character string supplied by the user which will be prepended to the error message of the last PVM call.
- `info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_perror` returns the error message of the last PVM call. The user can use `msg` to add additional information to the error message, for example, its location.

All stdout and stderr messages are placed in the file /tmp/pvml.<uid> on the master pvmd's host.

Examples

C:

```
if( pvm_send( tid, msgtag );
  pvm_perror();
```

Fortran:

```
CALL PVMFSEND( TID, MSGTAG )
IF( INFO .LT. 0 ) CALL PVMFPERROR( 'Step 6', INFO )
```

Errors

No error condition is returned by `pvm_perror`.

pvmfprecv() pvm_precv()

Receive a message directly into a buffer.

Synopsis

C	`int info = pvm_psend( int tid, int msgtag,`
	`char *buf, int len, int datatype )`
	`int atid, int atag, int alen )`

Fortran	`call pvmfpsend( tid, msgtag, buf, len, datatype,` `atid, atag, alen, info )`

Parameters

`tid`	–	integer task identifier of sending process (to match).
`msgtag`	–	integer message tag (to match) msgtag should be >= 0.
`buf`	–	Pointer to a buffer to receive into.
`len`	–	Length of buffer (in multiple of data type size).
`datatype`	–	Type of data to which buf points (see below).
`atid`	–	Returns actual TID of sender.
`atag`	–	Returns actual message tag.
`atid`	–	Returns actual message length.
`info`	–	integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_precv` blocks the process until a message with label `msgtag` has arrived from `tid`. pvm_precv then places the contents of the message in the supplied buffer, `buf`, up to a maximum length of len * (size of data type).

pvm_precv can receive messages sent by pvm_psend, pvm_send, pvm_mcast, or pvm_bcast.

A -1 in `msgtag` or `tid` matches anything. This allows the user the following options. If tid $= -1$ and msgtag is defined by the user, then pvm_precv will accept a message from any process which has a matching msgtag. If msgtag $= -1$ and tid is defined by the user, then pvm_precv will accept any message that is sent from process tid. If tid $= -1$ and msgtag $= -1$, then pvm_precv will accept any message from any process.

In C the datatype parameter must be one of the following, depending on the type of data to be sent:

datatype	Data Type
PVM_STR	string

PVM_BYTE	byte
PVM_SHORT	short
PVM_INT	int
PVM_FLOAT	real
PVM_CPLX	complex
PVM_DOUBLE	double
PVM_DCPLX	double complex
PVM_LONG	long integer
PVM_USHORT	unsigned short int
PVM_UINT	unsigned int
PVM_ULONG	unsigned long int

In Fortran the same data types specified for pvmfunpack() should be used.

The PVM model guarantees the following about message order. 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.

pvm_precv is blocking which means the routine waits until a message matching the user specified tid and msgtag values arrives at the local pvmd. If the message has already arrived then pvm_recv returns immediately with the message.

pvm_precv does not affect the state of the current receive message buffer (created by the other receive functions).

Examples

C:

```
info = pvm_precv( tid, msgtag, array, cnt, PVM_FLOAT,
                  &src, &atag, &acnt );
```

Fortran:

```
CALL PVMFPRECV( -1, 4, BUF, CNT, REAL4,
                SRC, ATAG, ACNT, INFO )
```

Errors

These error conditions can be returned by `pvm_prec`

Name	Possible cause
PvmBadParam	giving an invalid tid or a msgtag.
PvmSysErr	pvmd not responding.

pvmfprobe() pvm_probe()

check if message has arrived

Synopsis

C	`int bufid = pvm_probe( int tid, int msgtag )`
Fortran	`call pvmfprobe( tid, msgtag, bufid )`

Parameters

`tid`	–	integer task identifier of sending process supplied by the user. (A -1 in this argument matches any tid (wildcard).)
`msgtag`	–	integer message tag supplied by the user. msgtag should be >= 0. (A -1 in this argument matches any message tag (wildcard).)
`bufid`	–	integer returning the value of the new active receive buffer identifier. Values less than zero indicate an error.

Discussion

The routine `pvm_probe` checks to see if a message with label `msgtag` has arrived from `tid`. If a matching message has arrived pvm_probe returns a buffer identifier in `bufid`. This `bufid` can be used in a pvm_bufinfo call to determine information about the message such as its source and length.

If the requested message has not arrived, then pvm_probe returns with a 0 in `bufid`. If some error occurs `bufid` will be < 0.

A -1 in `msgtag` or `tid` matches anything. This allows the user the following options. If tid $= -1$ and msgtag is defined by the user, then pvm_probe will accept a message from any process which has a matching msgtag. If msgtag $= -1$ and tid is defined by the user, then pvm_probe will accept any message that is sent from process tid. If tid $= -1$ and msgtag $= -1$, then pvm_probe will accept any message from any process.

pvm_probe can be called multiple times to check if a given message has arrived yet. After the message has arrived, pvm_recv must be called before the message can be unpacked into the user's memory using the unpack routines.

Examples

C:

```
tid = pvm_parent();
msgtag = 4 ;
arrived = pvm_probe( tid, msgtag );
if( arrived > 0 )
  info = pvm_bufinfo( arrived, &len, &tag, &tid );
else
/* go do other computing */
```

Fortran:

```
          CALL PVMFPROBE( -1, 4, ARRIVED )
          IF ( ARRIVED .GT. 0 ) THEN
             CALL PVMFBUFINFO( ARRIVED, LEN, TAG, TID, INFO )
          ELSE
*            GO DO USEFUL WORK
          ENDIF
```

Errors

These error conditions can be returned by `pvm_probe`.

Name	Possible cause
PvmBadParam	giving an invalid tid value or msgtag.
PvmSysErr	pvmd not responding.

pvmfpsend() pvm_psend()

Pack and send data in one call.

Synopsis

C	`int info = pvm_psend( int tid, int msgtag, char *buf, int len, int datatype )`
Fortran	`call pvmfpsend( tid, msgtag, buf, len, datatype, info )`

Parameters

`tid`	–	integer task identifier of destination process.
`msgtag`	–	integer message tag supplied by the user. msgtag should be >= 0.
`buf`	–	Pointer to a buffer to send.
`len`	–	Length of buffer (in multiple of data type size).
`datatype`	–	Type of data to which buf points (see below).
`info`	–	integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_psend` takes a pointer to a buffer `buf` its length `len` and its data type `datatype` and sends this data directly to the PVM task identified by `tid`.

pvm_psend data can be received by pvm_precv, pvm_recv, pvm_trecv, or pvm_nrecv.

`msgtag` is used to label the content of the message. If pvm_psend is successful, `info` will be 0. If some error occurs then `info` will be < 0.

The pvm_psend routine is asynchronous. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processor. This is in contrast to synchronous communication, during which computation on the sending processor halts until the matching receive is executed by the receiving processor.

In C the datatype parameter must be one of the following, depending on the type of data to be sent:

datatype	Data Type
PVM_STR	string
PVM_BYTE	byte
PVM_SHORT	short
PVM_INT	int
PVM_FLOAT	real
PVM_CPLX	complex
PVM_DOUBLE	double

```
PVM_DCPLX       double complex
PVM_LONG        long integer
PVM_USHORT      unsigned short int
PVM_UINT        unsigned int
PVM_ULONG       unsigned long int
```

In Fortran the same data types specified for pack should be used.

The PVM model guarantees the following about message order. 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.

pvm_psend does not affect the state of the current outgoing message buffer (created by pvm_initsend and used by pvm_send).

Examples

C:

```
info = pvm_psend( tid, msgtag, array, 1000, PVM_FLOAT );
```

Fortran:

```
CALL PVMFPSEND( TID, MSGTAG, BUF, CNT, REAL4, INFO )
```

Errors

These error conditions can be returned by `pvm_psend`

Name	Possible cause
PvmBadParam	giving an invalid tid or a msgtag.
PvmSysErr	pvmd not responding.

pvmfpstat() pvm_pstat()

returns the status of the specified PVM process.

Synopsis

C	`int status = pvm_pstat( tid )`
Fortran	`call pvmfpstat( tid, status )`

Parameters

`tid` – integer task identifier of the PVM process in question.

`status` – integer returns the status of the PVM process identified by `tid`. Status is PvmOk if the task is running, PvmNoTask if not, and PvmBadParam if the `tid` is bad.

Discussion

The routine `pvm_pstat` returns the status of the process identified by `tid`. Also note that pvm_notify() can be used to notify the caller that a task has failed.

Examples

C:

```
tid = pvm_parent();
status = pvm_pstat( tid );
```

Fortran:

```
 CALL PVMFPARENT( TID )
 CALL PVMFPSTAT( TID, STATUS )
```

Errors

The following error conditions can be returned by `pvm_pstat`.

Name	Possible cause
PvmBadParam	Bad Parameter most likely an invalid tid value.
PvmSysErr	pvmd not responding.
PvmNoTask	Task not running.

pvmfrecv() pvm_recv()

receive a message

Synopsis

C	`int bufid = pvm_recv( int tid, int msgtag )`
Fortran	`call pvmfrecv( tid, msgtag, bufid )`

Parameters

`tid` – integer task identifier of sending process supplied by the user. (A -1 in this argument matches any tid (wildcard).)

`msgtag` – integer message tag supplied by the user. msgtag should be >= 0. It allows the user's program to distinguish between different kinds of messages . (A -1 in this argument matches any message tag (wildcard).)

`bufid` – integer returns the value of the new active receive buffer identifier. Values less than zero indicate an error.

Discussion

The routine `pvm_recv` blocks the process until a message with label `msgtag` has arrived from `tid`. pvm_recv then places the message in a new *active* receive buffer, which also clears the current receive buffer.

A -1 in `msgtag` or `tid` matches anything. This allows the user the following options. If tid $= -1$ and msgtag is defined by the user, then pvm_recv will accept a message from any process which has a matching msgtag. If msgtag $= -1$ and tid is defined by the user, then pvm_recv will accept any message that is sent from process tid. If tid $= -1$ and msgtag $= -1$, then pvm_recv will accept any message from any process.

The PVM model guarantees the following about message order. 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.

If pvm_recv is successful, `bufid` will be the value of the new active receive buffer identifier. If some error occurs then `bufid` will be < 0.

pvm_recv is blocking which means the routine waits until a message matching the user specified tid and msgtag values arrives at the local pvmd. If the message has already arrived then pvm_recv returns immediately with the message.

Once pvm_recv returns, the data in the message can be unpacked into the user's memory using the unpack routines.

Examples

C:

```
tid = pvm_parent();
msgtag = 4 ;
bufid = pvm_recv( tid, msgtag );
info = pvm_upkint( tid_array, 10, 1 );
info = pvm_upkint( problem_size, 1, 1 );
info = pvm_upkfloat( input_array, 100, 1 );
```

Fortran:

```
CALL PVMFRECV( -1, 4, BUFID )
CALL PVMFUNPACK( INTEGER4, TIDS, 25, 1, INFO )
CALL PVMFUNPACK( REAL8, MATRIX, 100, 100, INFO )
```

Errors

These error conditions can be returned by `pvm_recv`

Name	Possible cause
PvmBadParam	giving an invalid tid value, or msgtag < -1.
PvmSysErr	pvmd not responding.

pvm_recvf()

redefines the comparison function used to accept messages.

Synopsis

C

```
int (*old)() = pvm_recvf( int (*new)( int bufid,
                                      int tid, int tag ))
```

Fortran NOT AVAILABLE

Parameters

- `tid` – integer task identifier of sending process supplied by the user.
- `tag` – integer message tag supplied by the user.
- `bufid` – integer message buffer identifier.

Discussion

The routine `pvm_recvf` defines the comparison function to be used by the pvm_recv and pvm_nrecv functions. It is available as a means to customize PVM message passing. pvm_recvf sets a user supplied comparison function to evaluate messages for receiving . The default comparison function evaluates the source and message tag associated with all incoming messages.

pvm_recvf is intended for sophisticated C programmers who understand the function of such routines (like signal) and who require a receive routine that can match on more complex message contexts than the default provides.

pvm_recvf returns 0 if the default matching function; otherwise, it returns the matching function. The matching function should return:

Value	Action taken
< 0	return immediately with this error code
0	do not pick this message
1	pick this message and do not scan the rest
> 1	pick this highest ranked message after scanning them all

Example: Implementing probe with recvf

```
#include "pvm3.h"

static int foundit = 0;

static int
foo_match(mid, tid, code)
        int mid;
        int tid;
        int code;
{
        int t, c, cc;

        if ((cc = pvm_bufinfo(mid, (int*)0, &c, &t)) < 0)
                return cc;
        if ((tid == -1 || tid == t)
        && (code == -1 || code == c))
                foundit = 1;
        return 0;
}
int
probe(src, code)
{
        int (*omatch)();
        int cc;

        omatch = pvm_recvf(foo_match);
        foundit = 0;
        if ((cc = pvm_nrecv(src, code)) < 0)
                return cc;
        pvm_recvf(omatch);
        return foundit;
}
```

Errors

No error conditions are returned by `pvm_recvf`

pvmfreduce() pvm_reduce()

Performs a reduce operation over members of the specified group.

Synopsis

C
```
int info = pvm_reduce( void (*func)(),
                 void *data, int count, int datatype,
                 int msgtag, char *group, int root)
```
Fortran
```
call pvmfreduce( func, data, count, datatype,
                  msgtag, group, root, info )
```

Parameters

`func`	–	Function which defines the operation performed on the global data. Predefined are PvmMax, PvmMin, PvmSum and PvmProduct. Users can define their own function.
`data`	–	Pointer to the starting address of an array of local values. On return, the data array on the root will be overwritten with the result of the reduce operation over the group.
`count`	–	integer specifying the number of elements in data array.
`datatype`	–	integer specifying the type of the entries in the data array.
`msgtag`	–	integer message tag supplied by the user. msgtag should be >= 0.
`group`	–	Character string group name of an existing group.
`root`	–	Integer instance number of group member who gets the result.
`info`	–	Integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_reduce()` performs global operations such as max, min, and sum over all the tasks in a group. All group members call pvm_reduce() with their local data, and the result of the reduction operation appears on the user specified root task `root`. The root task is identified by its instance number in the group.

The pvm supplies the following predefined global functions that can be specified in `func`.

```
PvmMin
PvmMax
PvmSum
PvmProduct
```

PvmMax and PvmMin are implemented for all the datatypes listed below. For complex values the minimum [maximum] is that complex pair with the minimum [maximum] modulus. PvmSum and PvmProduct are implemented for for all the datatypes listed below with the exception of PVM_BYTE and BYTE1.

C and Fortran defined `datatypes` are:

```
 C datatypes   FORTRAN datatypes
-----------------------------------
 PVM_BYTE       BYTE1
 PVM_SHORT      INTEGER2
 PVM_INT        INTEGER4
 PVM_FLOAT      REAL4
 PVM_CPLX       COMPLEX8
 PVM_DOUBLE     REAL8
 PVM_DCPLX      COMPLEX16
 PVM_LONG
```

A user defined function may be used used in `func`.

SYNOPSIS for func

```
C   void func(int *datatype, void *x, void *y,
              int *num, int *info)

Fortran    call func(datatype, x, y, num, info)
```

func is the base function used for the reduction operation. Both x and y are arrays of type specified by datatype with num entries. The arguments datatype and info are as specified above. The arguments x and num correspond to data and count above. The argument y contains received values.

Note: pvm_reduce() does not block. if a task calls pvm_reduce and then leaves the group before the root has called pvm_reduce an error may occur.

The current algorithm is very simple and robust. A future implementation may make more efficient use of the architecture to allow greater parallelism.

Examples

C:

```
info =  pvm_reduce(PvmMax, &myvals, 10, PVM_INT,
                   msgtag, "workers", roottid);
```

Fortran:

```
CALL PVMFREDUCE(PvmMax, MYVALS, COUNT, INTEGER4,
                MTAG, 'workers', ROOT, INFO)
```

Errors

The following error conditions can be returned by `pvm_reduce`

Name	Possible cause
PvmBadParam	giving an invalid argument value.
PvmNoInst	Calling task is not in the group.
PvmSysErr	local pvmd is not responding.

——— pvm_reg_hoster()

Register this task as responsible for adding new PVM hosts.

Synopsis

C `#include <pvmsdpro.h>`
`int info = pvm_reg_hoster()`

Parameters

`info` – integer status code returned by the routine.

Discussion

The routine `pvm_reg_hoster` registers the calling task as a PVM slave pvmd starter. When the master pvmd receives a DM_ADD message, instead of starting the new slave pvmd processes itself, it passes a message to the hoster, which does the dirty work and sends a message back to the pvmd.

Note: This function isn't for beginners. If you don't grok what it does, you probably don't need it.

For a more complete explanation of what's going on here, you should refer to the PVM source code and/or user guide section on implementation; this is just a man page. That said...

When the master pvmd receives a `DM_ADD` message (request to add hosts to the virtual machine), it looks up the new host IP addresses, gets parameters from the host file if it was started with one, and sets default parameters. It then either attempts to start the processes (using *rsh* or *rexec()*) or, if a hoster has registered, sends it a `SM_STHOST` message.

The format of the SM_STHOST message is:

```
int nhosts // number of hosts
{
int tid // of host
string options // from hostfile so= field
string login // in form ``[username@]hostname.domain''
string command // to run on remote host
} [nhosts]
```

The hoster should attempt to run each command on each host and record the result. A command usually looks like:

```
$PVM_ROOT/lib/pvmd -s -d8 -nhonk 1 80a9ca95:0f5a 4096 3 80a95c43:0000
```

and a reply from a slave pvmd like:

```
ddpro<2312> arch<ALPHA> ip<80a95c43:0b3f> mtu<4096>
```

When finished, the hoster should send a `SM_STHOSTACK` message back to the address of the sender (the master pvmd). The format of the reply message is:

```
{
int tid // of host, must match request
string status // result line from slave or error code
} [] // implied count
```

The TIDs in the reply must match those in the request. They may be in a different order, however.

The result string should contain the entire reply (a single line) from each new slave pvmd, or an error code if something went wrong. Legal error codes are the literal names of the pvm_errno codes, for example "PvmCantStart". The default PVM hoster can return PvmDSysErr or PvmCantStart, and the slave pvmd itself can return PvmDupHost.

The hoster task must use pvm_setopt(PvmResvTids, 1) to allow sending reserved messages. Messages must be packed using data format PvmDataFoo.

pvm_reg_rm()

Register this task as PVM resource manager.

Synopsis

C
```
#include <pvmsdpro.h>
int info = pvm_reg_rm( struct hostinfo **hip )
struct hostinfo{
    int  hi_tid;
    char *hi_name;
    char *hi_arch;
    int  hi_speed;
} hip;
```

Parameters

`hostp` – pointer to an array of structures which contain information about each host including its pvmd task ID, name, architecture, and relative speed.

`info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_reg_rm()` registers the calling task as a PVM task and slave host scheduler. This means it intercepts certain libpvm calls from other tasks in order to have a say in scheduling policy. The scheduler will asynchronously receive messages from tasks containing requests for service, as well as messages from pvmds notifying it of system failures.

Before you start using this function, be warned that it's not a trivial thing, i.e. you can't just call it to turn off the default round-robin task assignment. Rather, it allows you to write your own scheduler and hook it to PVM.

To understand what the following messages mean, you should refer to the PVM source code and/or user guide section on implementation; There's just too much to say about them.

When one of the following libpvm functions is called in a task with resource manager set, the given message tag is sent to the scheduler.

Libpvm call	Sched. message	Normal message
pvm_addhosts()	SM_ADDHOST	TM_ADDHOST
pvm_config()	SM_CONFIG	TM_CONFIG
pvm_delhosts()	SM_DELHOST	TM_DELHOST
pvm_notify()	SM_NOTIFY	TM_NOTIFY
pvm_spawn()	SM_SPAWN	TM_SPAWN
pvm_tasks()	SM_TASK	TM_TASK
pvm_reg_sched()	SM_SCHED	TM_SCHED

The resource manager must in turn compose the following messages and send them to the pvmds:

Sched. message	Normal message
SM_EXEC	DM_EXEC
SM_EXECACK	DM_EXECACK
SM_ADD	DM_ADD
SM_ADDACK	DM_ADDACK
SM_HANDOFF	(none)

The following messages are sent asynchronously to the resource manager by the system:

Sched. message	Meaning
SM_TASKX	notify of task exit/fail
SM_HOSTX	notify of host delete/fail

The resource manager task must use pvm_setopt(PvmResvTids, 1) to allow sending reserved messages. Messages must be packed using data format PvmDataFoo.

——— **pvm_reg_tasker()**

Register this task as responsible for starting new PVM tasks.

Synopsis

C `#include <pvmsdpro.h>`
`int info = pvm_reg_tasker()`

Parameters

`info` – integer status code returned by the routine.

Discussion

The routine `pvm_reg_tasker` registers the calling task as a PVM task starter. When a tasker is registered with a pvmd, and the pvmd receives a DM_EXEC message, instead of fork()ing and exec()ing the task itself, it passes a message to the tasker, which does the dirty work and sends a message back to the pvmd.

Note: If this doesn't make sense, don't worry about it. This function is for folks who are writing stuff like debugger servers and so on. For a more complete explanation of what's going on here, you should refer to the PVM source code and/or user guide section on implementation; this is only a man page. That said...

When the pvmd receives a DM_EXEC message (request to exec new tasks), it searches *epath* (the PVM executable search path) for the file name. If it finds the file, it then either attempts to start the processes (using fork() and exec()) or, if a tasker has registered, sends it a SM_STTASK message.

The format of the SM_STTASK message is:

```
int tid // of task
int flags // as passed to spawn()
string path // absolute path of the executable
int argc // number of args to process
string argv[argc] // args
int nenv // number of envars to pass to task
string env[nenv] // environment strings
```

The tasker must attempt to start the process when it gets one of these messages. The tasker doesn't reply to the pvmd if the task is successfully started; the task will reconnect to the pvmd on its own (using the identifier in envar PVMEPID).

The tasker must send a SM_TASKX message to the pvmd when any task that it owns (has started) exits, or if it can't start a particular task. The format of the SM_TASKX message is:

```
int tid // of task
int status // the Unix exit status (from Iwait())
int u_sec // user time used by the task, seconds
int u_usec // microseconds
int s_sec // system time used by the task, seconds
int s_usec // microseconds
```

The tasker task must use pvm_setopt(PvmResvTids, 1) to allow sending reserved messages. Messages must be packed using data format PvmDataFoo.

pvmfscatter() pvm_scatter()

one group member sends a different portion of an array to each group member.

Synopsis

C
```
int info = pvm_scatter( void *result,  void *data,
                        int count, int datatype, int msgtag,
                        char *group, int rootginst)
```
Fortran
```
call pvmfscatter(result, data, count, datatype,
msgtag, group, rootginst, info)
```

Parameters

`result`	–	Pointer to the starting address of an array of length `count` of `datatype`
`data`	–	On the root this is a pointer to the starting address of an array `datatype` of local values which are to be accumulated from the members of the group. This array should be of length at least equal to the number of group members. times `count`. This argument is significant only on the root.
`count`	–	Integer specifying the number of array elements to be sent to each member of the group from the root.
`datatype`	–	Integer specifying the type of the entries in the result and data arrays. For a list of supported types see pvm_psend().
`msgtag`	–	Integer message tag supplied by the user. msgtag should be >= 0.
`group`	–	Character string group name of an existing group.
`rootginst`	–	Integer instance number of group member who performs the gather of the messages from the members of the group.
`info`	–	Integer status code returned by the routine. Values less than zero indicate an error.

Discussion

pvm_scatter() performs a scatter of data from the specified root member of the group to each of the members of the group, including itself. All group members must call pvm_scatter(), and each receives a portion of the `data` array from the root in their local `result` array. pvm_scatter() is the inverse of pvm_gather(). The first count entries in the root data array are sent to group member 1, the next count entries to group member 2, and so on.

In using the scatter and gather routines, keep in mind that C stores multidimensional arrays in row order, typically starting with an initial index of 0; whereas, Fortran stores arrays in column order, typically starting with an index of 1.

The current algorithm is very simple and robust. Future implementations will make more efficient use of the architecture to allow greater parallelism.

Examples

C:

```
info =  pvm_scatter(&getmyrow, &matrix, 10, PVM_INT,
                msgtag, "workers", rootginst);
```

Fortran:

```
CALL PVMFSCATTER(GETMYCOLUMN, MATRIX, COUNT, INTEGER4,
        MTAG, 'workers', ROOT, INFO)
```

Errors

These error conditions can be returned by `pvm_scatter`

Name	Possible cause
PvmBadParam	giving an invalid argument value.
PvmNoInst	Calling task is not in the group.
PvmSysErr	local pvmd is not responding.

pvmfsend() pvm_send()

sends the data in the active message buffer.

Synopsis

C	`int info = pvm_send( int tid, int msgtag )`
Fortran	`call pvmfsend( tid, msgtag, info )`

Parameters

`tid`	–	integer task identifier of destination process.
`msgtag`	–	integer message tag supplied by the user. msgtag should be >= 0.
`info`	–	integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_send` sends a message stored in the active send buffer to the PVM process identified by `tid`. `msgtag` is used to label the content of the message. If pvm_send is successful, `info` will be 0. If some error occurs then `info` will be < 0.

The pvm_send routine is asynchronous. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processor. This is in contrast to synchronous communication, during which computation on the sending processor halts until the matching receive is executed by the receiving processor.

pvm_send first checks to see if the destination is on the same machine. If so and this host is a multiprocessor then the vendor's underlying message passing routines are used to move the data between processes.

Examples

C:

```
info = pvm_initsend( PvmDataDefault );
info = pvm_pkint( array, 10, 1 );
msgtag = 3 ;
info = pvm_send( tid, msgtag );
```

Fortran:

```
CALL PVMFINITSEND(PVMRAW, INFO)
CALL PVMFPACK( REAL8, DATA, 100, 1, INFO )
CALL PVMFSEND( TID, 3, INFO )
```

Errors

These error conditions can be returned by `pvm_send`

Name	Possible cause
PvmBadParam	giving an invalid tid or a msgtag.
PvmSysErr	pvmd not responding.
PvmNoBuf	no active send buffer. Try calling pvm_initsend() before sending.

pvmfsendsig() pvm_sendsig()

sends a signal to another PVM process

Synopsis

C	`int info = pvm_sendsig( int tid, int signum )`
Fortran	`call pvmfsendsig( tid, signum, info )`

Parameters

`tid`	–	integer task identifier of PVM process to receive the signal.
`signum`	–	integer signal number.
`info`	–	integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_sendsig` sends the signal number `signum` to the PVM process identified by `tid`. If pvm_sendsig is successful, `info` will be 0. If some error occurs then `info` will be < 0.

pvm_sendsig should only be used by programmers with signal handling experience. It is very easy in a parallel environment for interrupts to cause nondeterministic behavior, deadlocks, and even system crashes. For example, if an interrupt is caught while a process is inside a Unix kernel call, then a graceful recovery may not be possible.

Examples

C:

```
 tid = pvm_parent();
info = pvm_sendsig( tid, SIGKILL);
```

Fortran:

```
CALL PVMFBUFINFO( BUFID, BYTES, TYPE, TID, INFO );
CALL PVMFSENDSIG( TID, SIGNUM, INFO )
```

Errors

These error conditions can be returned by `pvm_sendsig`

Name	Possible cause
PvmSysErr	pvmd not responding.
PvmBadParam	giving an invalid tid value.

pvmfsetopt() pvm_setopt()

Sets various libpvm options

Synopsis

C `int oldval = pvm_setopt( int what, int val )`
Fortran `call pvmfsetopt( what, val, oldval )`

Parameters

`what` – Integer defining what is being set. Options include:

Option	value	MEANING
`PvmRoute`	1	routing policy
`PvmDebugMask`	2	debugmask
`PvmAutoErr`	3	auto error reporting
`PvmOutputTid`	4	stdout device for children
`PvmOutputCode`	5	output msgtag
`PvmTraceTid`	6	trace device for children
`PvmTraceCode`	7	trace msgtag
`PvmFragSize`	8	message fragment size
`PvmResvTids`	9	Allow messages to reserved tags and TIDs
`PvmSelfOutputTid`	10	Stdout destination
`PvmSelfOutputCode`	11	Output message tag
`PvmSelfTraceTid`	12	Trace data destination
`PvmSelfTraceCode`	13	Trace message tag

`val` – Integer specifying new setting of option. Predefined route values are:

Option	value	MEANING
`PvmDontRoute`	1	
`PvmAllowDirect`	2	
`PvmRouteDirect`	3	

`oldval` – Integer returning the previous setting of the option.

Discussion

The routine `pvm_setopt` is a general purpose function to allow the user to set options in the PVM system. In PVM 3.2 pvm_setopt can be used to set several options including: automatic error message printing, debugging level, and communication routing method for all subsequent PVM calls. Pvm_setopt returns the previous value of set in oldval.

PvmRoute: In the case of communication routing, pvm_setopt advises PVM on whether or not to set up direct task-to-task links `PvmRouteDirect` (using TCP) for all subsequent communication. Once a link is established it remains until the application finishes. If a direct link can not be established because one of the two tasks has requested `PvmDontRoute` or because no resources are available, then the default route through the PVM daemons is used. On multiprocessors such as Intel Paragon this option is ignored because the communication between tasks on these machines always uses the native protocol for direct communication. pvm_setopt can be called multiple times to selectively establish direct links, but is typically set only once near the beginning of each task. `PvmAllowDirect` is the default route setting. This setting on task A allows other tasks to set up direct links to A. Once a direct link is established between tasks both tasks will use it for sending messages.

PvmDebugMask: For this option val is the debugging level. When debugging is turned on, PVM will log detailed information about its operations and progress on it's stderr stream. Default is no debug information.

PvmAutoErr: In the case of automatic error printing, Any PVM routines that return an error condition will automatically print the associated error message. The argument val defines whether this reporting is to be turned on (1) or turned off (0) for subsequent calls. A value of (2) will cause the program to exit after printing the error message (Not implemented in 3.2). Default is reporting turned on.

PvmOutputTid: For this option val is the stdout device for children. All the standard output from the calling task and any tasks it spawns will be redirected to the specified device. Val is the tid of a PVM task or pvmd. The Default val of 0 redirects stdout to master host, which writes to the log file /tmp/pvml.xxxx, where xxxx is the uid of the user.

PvmOutputCode: Only meaningful on task with PvmOutputTid set to itself. This is the message tag value to be used in receiving messages containing standard output from other tasks.

PvmTraceTid: For this option val is the task responsible for writing out trace event for the calling task and all its children. Val is the tid of a PVM task or pvmd. The Default val of 0 redirects trace to master host.

PvmTraceCode: Only meaningful on task with PvmTraceTid set to itself. This is the message tag value to be used in receiving messages containing trace output from other tasks.

PvmFragSize: For this option val specifies the message fragment size in bytes. Default value varies with host architecture.

PvmResvTids: A val of 1 enables the task to send messages with reserved tags and to non-task destinations. The default (0) results in a PvmBadParam error instead.

PvmSelfOutputTid: Sets the Istdout destination for the task. Everything printed on stdout is packed into messages and sent to the destination. Note: this only works for spawned tasks, because the pvmd doesn't get the output from tasks started by other means. val is the TID of a PVM task. Setting PvmSelfOutputTid to 0 redirects stdout to the master pvmd, which writes to the log file /tmp/pvml.xxxx. The default setting is inherited from the parent task, else is 0. Setting either PvmSelfOutputTid or PvmSelfOutputCode also causes both PvmOutputTid and PvmOutputCode to take on the values of PvmSelfOutputTid and PvmSelfOutputCode, respectively.

PvmSelfOutputCode Sets the message tag for standard output messages.

PvmSelfTraceTid Sets the trace data message destination for the task. Libpvm trace data is sent as messages to the destination. val is the TID of a PVM task. Setting PvmSelfTraceTid to 0 discards trace data. The default setting is inherited from the parent task, else is 0. Setting either PvmSelfTraceTid or PvmSelfTraceCode also causes both PvmTraceTid and PvmTraceCode to take on the values of PvmSelfTraceTid and PvmSelfTraceCode, respectively.

PvmSelfTraceCode Sets the message tag for trace data messages.

pvm_setopt returns the previous value of the option.

Examples

C:

```
oldval = pvm_setopt( PvmRoute, PvmRouteDirect );
```

Fortran:

```
CALL PVMFSETOPT( PVMAUTOERR, 1, OLDVAL )
```

Errors

These error conditions can be returned by `pvm_setopt`

Name	Possible cause
PvmBadParam	giving an invalid arg.

pvmfsetrbuf() pvm_setrbuf()

switches the active receive buffer and saves the previous buffer.

Synopsis

C	`int oldbuf = pvm_setrbuf( int bufid )`
Fortran	`call pvmfsetrbuf( bufid, oldbuf )`

Parameters

`bufid`	–	integer specifying the message buffer identifier for the new active receive buffer.
`oldbuf`	–	integer returning the message buffer identifier for the previous active receive buffer.

Discussion

The routine `pvm_setrbuf` switches the active receive buffer to `bufid` and saves the previous active receive buffer `oldbuf`. If `bufid` is set to 0 then the present active receive buffer is saved and no active receive buffer exists.

A successful receive automatically creates a new active receive buffer. If a previous receive has not been unpacked and needs to be saved for later, then the previous bufid can be saved and reset later to the active buffer for unpacking.

The routine is required when managing multiple message buffers. For example switching back and forth between two buffers. One buffer could be used to send information to a graphical interface while a second buffer could be used send data to other tasks in the application.

Examples

C:

```
rbuf1 = pvm_setrbuf( rbuf2 );
```

Fortran:

```
CALL PVMFSETRBUF( NEWBUF, OLDBUF )
```

Errors

These error conditions can be returned by `pvm_setrbuf`

Name	Possible cause
PvmBadParam	giving an invalid bufid.
PvmNoSuchBuf	switching to a non-existent message buffer.

pvmfsetsbuf() pvm_setsbuf()

switches the active send buffer.

Synopsis

C	`int oldbuf = pvm_setsbuf( int bufid )`
Fortran	`call pvmfsetsbuf( bufid, oldbuf )`

Parameters

`bufid`	–	integer message buffer identifier for the new active send buffer. A value of 0 indicates the default receive buffer.
`oldbuf`	–	integer returning the message buffer identifier for the previous active send buffer.

Discussion

The routine `pvm_setsbuf` switches the active send buffer to `bufid` and saves the previous active send buffer `oldbuf`. If `bufid` is set to 0 then the present active send buffer is saved and no active send buffer exists.

The routine is required when managing multiple message buffers. For example switching back and forth between two buffers. One buffer could be used to send information to a graphical interface while a second buffer could be used send data to other tasks in the application.

Examples

C:

```
sbuf1 = pvm_setsbuf( sbuf2 );
```

Fortran:

```
CALL PVMFSETSBUF( NEWBUF, OLDBUF )
```

Errors

These error conditions can be returned by `pvm_setsbuf`

Name	Possible cause
PvmBadParam	giving an invalid bufid.
PvmNoSuchBuf	switching to a non-existent message buffer.

pvmfspawn() pvm_spawn()

starts new PVM processes.

Synopsis

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

Fortran
```
call pvmfspawn( task, flag, where,
                ntask, tids, numt )
```

Parameters

`task` – character string containing the executable file name of the PVM process to be started. The executable must already reside on the host on which it is to be started. The default location PVM looks is `$HOME/pvm3/bin/$PVM_ARCH/filename` .

`argv` – pointer to an array of arguments to the executable with the end of the array specified by NULL. If the executable takes no arguments, then the second argument to pvm_spawn is NULL.

`flag` – integer specifying spawn options.

In C `flag` should be the **sum** of:

Option value		MEANING
`PvmTaskDefault`	0	PVM can choose any machine to start task
`PvmTaskHost`	1	`where` specifies a particular host
`PvmTaskArch`	2	`where` specifies a type of architecture
`PvmTaskDebug`	4	start up processes under debugger
`PvmTaskTrace`	8	processes will generate PVM trace data. *
`PvmMppFront`	16	Start process on MPP front-end.
`PvmHostCompl`	32	Use complement host set

`where` – character string specifying where to start the PVM process. Depending on the value of `flag`, `where` can be a host name such as "ibm1.epm.ornl.gov" or a PVM architecture class such as "SUN4". If `flag` is 0, then `where` is ignored and PVM will select the most appropriate host.

`ntask` – integer specifying the number of copies of the executable to start up.

`tids` – integer array of length at least `ntask`. On return the array contains the tids of the PVM processes started by this pvm_spawn call. If there is a error starting a given task, then that location in the array will contain the associated error code.

`numt` – integer returning the actual number of tasks started. Values less than zero indicate a system error. A positive value less than `ntask` indicates a partial failure. In this case the user should check the `tids` array for the error code(s).

Discussion

The routine `pvm_spawn` starts up `ntask` copies of the executable named `task`. On systems that support environment, spawn passes exported variables in the parent environment to children tasks. If set, the envar PVM_EXPORT is passed and if PVM_EXPORT contains other names (separated by ':') they will be passed too. this is useful for e.g.:

```
setenv DISPLAY myworkstation:0.0
setenv MYSTERYVAR 13
setenv PVM_EXPORT DISPLAY:MYSTERYVAR
```

The hosts on which the PVM processes are started is set by the `flag` and `where` arguments. On return the array `tids` contains the PVM task identifiers for each process started.

If pvm_spawn starts one or more tasks, `numt` will be the actual number of tasks started. If a system error occurs then `numt` will be < 0, If `numt` is less than `ntask` then some executables have failed to start and the user should check the last `ntask` - `numt` locations in the `tids` array which will contain the associated error codes, see below for meaning. Meaning the first numt tids in the array are good, which can be useful for functions such as pvm_mcast().

When `flag` is set to 0 and `where` is set to NULL (or "*" in Fortran) a heuristic is used to distribute the `ntask` processes across the virtual machine. Initially the heuristic is round-robin assignment starting with the next host in the table. Later PVM will use the metrics of machine load and rated performance (sp=) to determine the most appropriate hosts.

If the PvmHostCompl flag is set, the resulting host set gets complemented. Also, the TaskHost hostname "." is taken as localhost. This allows spawning tasks on

"." to get the localhost or to spawn n - 1 things on TaskHost—HostCompl "." to get any but the localhost.

In the special case where a multiprocessor is specified by `where`, pvm_spawn will start all `ntask` copies on this single machine using the vendor's underlying routines.

If PvmTaskDebug is set, then the pvmd will start the task(s) in a debugger. In this case, instead of executing `pvm3/bin/ARCH/task args` it executes `pvm3/lib/debugger pvm3/bin/ARCH/task args`. Debugger is a shell script that the users can modify to their individual tastes. Presently the script starts an xterm with dbx or comparable debugger in it.

Examples

C:

```
numt = pvm_spawn( "host", 0, PvmTaskHost, "sparky", 1, &tid[0] );
numt = pvm_spawn( "host", 0, (PvmTaskHost+PvmTaskDebug),
                  "sparky", 1, &tid[0] );
numt = pvm_spawn( "node", 0, PvmTaskArch, "RIOS", 1, &tid[i] );
numt = pvm_spawn( "FEM1", args, 0, 0, 16, tids );
numt = pvm_spawn( "pde", 0, PvmTaskHost, "paragon.ornl", 512, tids );
```

Fortran:

```
FLAG = PVMARCH + PVMDEBUG
CALL PVMFSPAWN( 'node', FLAG, 'SUN4', 1, TID(3), NUMT )
CALL PVMFSPAWN( 'FEM1', PVMDEFAULT, '*', 16, TIDS, NUMT )
CALL PVMFSPAWN( 'TBMD', PVMHOST, 'cm5.utk.edu', 32, TIDS, NUMT )
```

Errors

These error conditions can be returned by `pvm_spawn` either in `numt` or in the `tids` array.

Name	Value	Possible cause
PvmBadParam	-2	giving an invalid argument value.
PvmNoHost	-6	Specified host is not in the virtual machine.
PvmNoFile	-7	Specified executable can not be found. The default location PVM looks in `~/pvm3/bin/ARCH` where ARCH is PVM architecture name.
PvmNoMem	-10	Malloc failed. Not enough memory on host.
PvmSysErr	-14	pvmd not responding.
PvmOutOfRes	-27	out of resources.

pvmftasks() pvm_tasks()

Returns information about the tasks running on the virtual machine.

Synopsis

C

```
int info = pvm_tasks( int where, int *ntask,
                      struct pvmtaskinfo **taskp )
struct pvmtaskinfo{
    int  ti_tid;
    int  ti_ptid;
    int  ti_host;
    int  ti_flag;
    char *ti_a_out;
    int  ti_pid;
} taskp;
```

Fortran

```
call pvmftasks( where, ntask, tid, ptid,
dtid, flag, aout,info )
```

Parameters

- `where` – integer specifying what tasks to return information about. The options are the following:

0	for all the tasks on the virtual machine
pvmd tid	for all tasks on a given host
tid	for a specific task

- `ntask` – integer returning the number of tasks being reported on.
- `taskp` – pointer to an array of structures which contain information about each task including its task ID, parent tid, pvmd task ID, status flag, the name of this task's executable file, and task (O/S dependent) process id. The status flag values are waiting for a message, waiting for the pvmd, and running.
- `tid` – integer returning task ID of one task
- `ptid` – integer returning parent task ID
- `dtid` – integer returning pvmd task ID of host task is on.
- `flag` – integer returning status of task
- `aout` – character string returning the name of spawned task. Manually started tasks return blank.
- `info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

The routine `pvm_tasks` returns information about tasks running on the virtual machine. The information returned is the same as that available from the console command `ps`. The C function returns information about the entire virtual machine in one call. The Fortran function returns information about one task per call and cycles through all the tasks. Thus, if where = 0, and pvmftasks is called ntask times, all tasks will be represented.

If pvm_tasks is successful, `info` will be 0. If some error occurs, `info` will be < 0.

Examples

C:

```
info = pvm_tasks( 0, &ntask, &taskp );
```

Fortran:

```
CALL PVMFTASKS( DTID, NTASK, INFO )
```

Errors

The following error conditions can be returned by `pvm_tasks`.

Name	Possible Cause
PvmBadParam	invalid value for `where` argument.
PvmSysErr	pvmd not responding.
PvmNoHost	specified host not in virtual machine.

pvmftidtohost() pvm_tidtohost()

returns the host ID on which the specified task is running.

Synopsis

C	`int dtid = pvm_tidtohost( int tid )`
Fortran	`call pvmftidtohost( tid, dtid )`

Parameters

`tid` – integer task identifier specified.

`dtid` – integer tid of the host's pvmd returned.

Discussion

The routine `pvm_tidtohost` returns the host ID `dtid` on which the specified task `tid` is running.

Examples

C:

```
host = pvm_tidtohost( tid[0] );
```

Fortran:

```
CALL PVMFTIDTOHOST(TID, HOSTID)
```

Errors

These error conditions can be returned by `pvm_tidtohost`:

Name	Possible cause
PvmBadParam	giving an invalid tid.

pvmftrecv() pvm_trecv()

receive with timeout.

Synopsis

C	`int bufid = pvm_trecv( int tid, int msgtag, struct timeval *tmout )`
Fortran	`call pvmftrecv( tid, msgtag, sec, usec, bufid )`

Parameters

`tid`	–	Integer to match task identifier of sending process.
`msgtag`	–	Integer to match message tag; should be ¿= 0.
`tmout`	–	Time to wait before returning without a message.
`sec, usec`	–	Integers defining Time to wait before returning without a message.
`bufid`	–	integer returns the value of the new active receive buffer identifier. Values less than zero indicate an error.

Discussion

The routine `pvm_trecv` blocks the process until a message with label `msgtag` has arrived from `tid`. pvm_trecv then places the message in a new *active* receive buffer, also clearing the current receive buffer. If no matching message arrives within the specified waiting time, pvm_trecv returns without a message.

A -1 in `msgtag` or `tid` matches anything. This allows the user the following options. If tid $= -1$ and msgtag is defined by the user, then pvm_recv will accept a message from any process which has a matching msgtag. If msgtag $= -1$ and tid is defined by the user, then pvm_recv will accept any message that is sent from process tid. If tid $= -1$ and msgtag $= -1$, then pvm_recv will accept any message from any process.

In C, the `tmout` fields tv_sec and tv_usec specify how long pvm_trecv will wait without returning a matching message. In Fortran, two separate parameters, `sec` and `usec` are passed. With both set to zero, pvm_trecv behaves the same as pvm_nrecv(), which is to probe for messages and return immediately even if none are matched. In C, passing a null pointer in tmout makes pvm_trecv act like pvm_recv(), that is, it will wait indefinitely. In Fortran, setting sec to -1 has the same effect.

The PVM model guarantees the following about message order. 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.

If pvm_trecv is successful, `bufid` will be the value of the new active receive buffer identifier. If some error occurs then `bufid` will be < 0.

Once pvm_trecv returns, the data in the message can be unpacked into the user's memory using the unpack routines.

Examples

C:

```
struct timeval tmout;

tid = pvm_parent();
msgtag = 4 ;
if ((bufid = pvm_trecv( tid, msgtag, &tmout )) >0) {
 pvm_upkint( tid_array, 10, 1 );
 pvm_upkint( problem_size, 1, 1 );
 pvm_upkfloat( input_array, 100, 1 );
}
```

Fortran:

```
CALL PVMFRECV( -1, 4, 60, 0, BUFID )
IF (BUFID .GT. 0) THEN
CALL PVMFUNPACK( INTEGER4, TIDS, 25, 1, INFO )
CALL PVMFUNPACK( REAL8, MATRIX, 100, 100, INFO )
ENDIF
```

Errors

These error conditions can be returned

Name	Possible cause
PvmBadParam	giving an invalid tid value, or msgtag < -1.
PvmSysErr	pvmd not responding.

pvmfunpack() pvm_upk*()

unpack the active message buffer into arrays of prescribed data type.

Synopsis

C

```
int info = pvm_unpackf( const char *fmt, ... )
int info = pvm_upkbyte(  char   *xp, int nitem, int stride )
int info = pvm_upkcplx(  float  *cp, int nitem, int stride )
int info = pvm_upkdcplx( double *zp, int nitem, int stride )
int info = pvm_upkdouble(double *dp, int nitem, int stride )
int info = pvm_upkfloat( float  *fp, int nitem, int stride )
int info = pvm_upkint(   int    *ip, int nitem, int stride )
int info = pvm_upklong(  long   *ip, int nitem, int stride )
int info = pvm_upkshort( short  *jp, int nitem, int stride )
int info = pvm_upkstr(   char   *sp )
```

Fortran

```
call pvmfunpack( what, xp, nitem, stride, info )
```

Parameters

- `fmt` – Printf-like format expression specifying what to pack. (See discussion)
- `nitem` – The total number of *items* to be unpacked (not the number of bytes).
- `stride` – The stride to be used when packing the items. For example, if stride= 2 in pvm_upkcplx, then every other complex number will be unpacked.
- `xp` – pointer to the beginning of a block of bytes. Can be any data type, but must match the corresponding pack data type.
- `cp` – complex array at least nitem*stride items long.
- `zp` – double precision complex array at least nitem*stride items long.
- `dp` – double precision real array at least nitem*stride items long.
- `fp` – real array at least nitem*stride items long.
- `ip` – integer array at least nitem*stride items long.
- `jp` – integer*2 array at least nitem*stride items long.
- `sp` – pointer to a null terminated character string.

`what` – integer specifying the type of data being unpacked.

`what` options			
STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

`info` – integer status code returned by the routine. Values less than zero indicate an error.

Discussion

Each of the `pvm_upk*` routines unpacks an array of the given data type from the active receive buffer. The arguments for each of the routines are a pointer to the array to be unpacked into, `nitem` which is the total number of items to unpack, and `stride` which is the stride to use when unpacking.

An exception is pvm_upkstr() which by definition unpacks a NULL terminated character string and thus does not need `nitem` or `stride` arguments. The Fortran routine pvmfunpack(STRING, ...) expects `nitem` to be the number of characters in the string and `stride` to be 1.

If the unpacking is successful, `info` will be 0. If some error occurs then `info` will be < 0.

A single variable (not an array) can be unpacked by setting `nitem`$= 1$ and `stride`$= 1$.

The routine pvm_unpackf() uses a printf-like format expression to specify what and how to unpack data from the receive buffer. All variables are passed as addresses. A BNF-like description of the format syntax is:

```
format : null | init | format fmt
init : null | '%' '+'
fmt : '%' count stride modifiers fchar
fchar : 'c' | 'd' | 'f' | 'x' | 's'
count : null | [0-9]+ | '*'
stride : null | '.' ( [0-9]+ | '*' )
modifiers : null | modifiers mchar
mchar : 'h' | 'l' | 'u'
```

```
Formats:
  +  means initsend - must match an int (how) in the param list.
  c  pack/unpack bytes
  d  integer
  f  float
  x  complex float
  s  string
```

```
Modifiers:
  h  short (int)
  l  long  (int, float, complex float)
  u  unsigned (int)

'*' count or stride must match an int in the param list.
```

Future extensions to the `what` argument will include 64 bit types when XDR encoding of these types is available. Meanwhile users should be aware that precision can be lost when passing data from a 64 bit machine like a Cray to a 32 bit machine like a SPARCstation. As a mnemonic the `what` argument name includes the number of bytes of precision to expect. By setting encoding to PVMRAW (see pvmfinitsend) data can be transferred between two 64 bit machines with full precision even if the PVM configuration is heterogeneous.

Messages should be unpacked exactly like they were packed to insure data integrity. Packing integers and unpacking them as floats will often fail because a type encoding will have occurred transferring the data between heterogeneous hosts. Packing 10 integers and 100 floats then trying to unpack only 3 integers and the 100 floats will also fail.

Examples

C:

```
info = pvm_recv( tid, msgtag );
info = pvm_upkstr( string );
info = pvm_upkint( &size, 1, 1 );
info = pvm_upkint( array, size, 1 );
info = pvm_upkdouble( matrix, size*size, 1 );
```

Fortran:

```
CALL PVMFRECV( TID, MSGTAG );
CALL PVMFUNPACK( INTEGER4, NSIZE, 1, 1, INFO )
CALL PVMFUNPACK( STRING, STEPNAME, 8, 1, INFO )
CALL PVMFUNPACK( REAL4, A(5,1), NSIZE, NSIZE , INFO )
```

Errors

Name	Possible cause
PvmNoData	Reading beyond the end of the receive buffer. Most likely cause is trying to unpack more items than were originally packed into the buffer.
PvmBadMsg	The received message can not be decoded. Most likely because the hosts are heterogeneous and the user specified an incompatible encoding. Try setting the encoding to PvmDataDefault (see pvm_mkbuf).
PvmNoBuf	There is no active receive buffer to unpack.