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: May, 1993

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

1 Introduction 1
2 Features in PVM 3 2
2.1 Updated User interface 2
2.2 Integer Task Identifier 2
2.3 Process Control 2
2.4 Fault Tolerance 3
2.5 Dynamic Process Groups 3
2.6 Signaling 3
2.7 Communication 3
2.8 Multiprocessor Integration 4
3 Getting and Installing PVM 4
3.1 Obtaining PVM 6
3.2 Unpacking 6
3.3 Building 6
3.4 Installing 7
4 Configuring and Starting PVM 7
4.1 Starting PVM 7
4.2 Configuring a Virtual Machine 8
4.3 Troubleshooting Startup 9
4.4 Compiling PVM Applications 11
4.5 Running PVM Applications 11
5 User Interface 12
5.1 Process Control 12
5.2 Information 14
5.3 Dynamic Configuration 15
5.4 Signaling 15
5.5 Error Messages 15
5.6 Message Passing 16
5.6.1 Message Buffers 16
5.6.2 Packing Data 18
5.6.3 Sending and Receiving Data 19
5.6.4 Unpacking Data 20
6 Dynamic Process Groups 21
7 Examples in C and Fortran 22
8 Writing Applications 23
8.1 General performance considerations 23
8.2 Network particular considerations 31
8.3 Load Balancing 32
9 Debugging Methods 33
10 Implementation Details 35

10.1 TID Details 36
10.2 Messages 37
10.3 Libpvm Internals 37
10.3.1 Connecting to the Pvmd 37
10.4 Pvmd Internals 38
10.4.1 Task Management 38
10.4.2 Pvmd Wait Context 38
10.4.3 Machine Reconfiguration 39
10.4.4 Pvmd Fault Detection 40
10.5 Multiprocessor Interface 40
10.6 Instrumentation 42
11 Support 43
12 Appendix A. Reference pages for PVM 3.0 routines 44
13 Appendix B. Mapping of PVM 2.4 routines to PVM 3.0 routines 104
14 Appendix C. PVM 3.0 Quick Installation Guide 107
15 Appendix D. PVM 3.0 Quick Reference Guide 108

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 3.1 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.

A PVM 3.1 quick reference card, a quick installation guide, a mapping of 2.4 routines to 3 routines, and a copy of the PVM 3.1 man pages are included in appendices.

`[This ORNL report is still in draft form, we plan to add examples and discussion to the dynamic groups section and more details about the PVM 3 internals.]`

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 3.1 options and error conditions as well as quick reference and installation procedures.

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, 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 executes pvmd3 on one of the computers which in turn starts up pvmd3 on each of the computers making up the user-defined virtual machine. The PVM application can then be started from a Unix prompt on any of

these computers. 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. The following sections describe the features that are available in PVM 3.

2.1. Updated User interface

The project team has gotten feedback from several users that 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 incorporate 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.x 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 previous versions 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(), 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.

2.4. Fault Tolerance

If a host fails, PVM will automatically detect this and deleted 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 set of tasks about an event by sending them a message with a user-specified tag that the application can check for. The notification events include 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 recieve, and non-blocking receive functions. In our terminology, a blocking send returns as soon as the send buffer is free for reuse regardless of the state of the receiver. 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 size messages 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. Distributed memory multicomputers such as the Intel iPSC/860 often have only a host or a special node that is actually connected to the network. All the other processors can only communicate over the internal communication paths. To exploit such machines with PVM 2.4 it was necessary to write a special PVM program that ran on the "host" processor and usually didn't participate in the computation. This program translated application messages back and forth between the PVM format and the native message-passing format and routed the messages using native or PVM routines as required.

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 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 3.1 so that Intel's NX message-passing routines are used for inter-node communication. Cray, Convex, SGI, DEC, KSR, and IBM are also supplying PVM 3 compatibility with their respective multiprocessors when they become available. More multiprocessor machines will be added to subsequent PVM 3.x releases.

3. Getting and Installing PVM

Only one person at an organization needs to get and install PVM for everyone at that organization to use it. *ARCH* is used throughout this report to represent the architecture name PVM uses for a given computer. Table 1 lists all the ARCH names and their corresponding architecture types that are supported in PVM 3.1.

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, Cray-2	UNICOS
CRAYSMP	Cray S-MP	
DGAV	Data General Aviion	
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
NEXT	NeXT	
PGON	Intel Paragon	link -lrpc
PMAX	DECstation 3100, 5100	Ultrix
RS6K	IBM/RS6000	AIX
RT	IBM RT	
SGI	Silicon Graphics IRIS	link -lsun
SUN3	Sun 3	SunOS
SUN4	Sun 4, SPARCstation	
SYMM	Sequent Symmetry	
TITN	Stardent Titan	
UVAX	DEC MicroVAX	

Table 1: ARCH names used in PVM 3.

3.1. Obtaining PVM

If PVM is not installed at your site, then you should obtain and install the PVM software. PVM does not require special privileges to be installed. Anyone with a valid login can do so, but it is a common courtesy to inform your system administrator of your intentions.

The second step is to obtain the software and documentation. This user's guide, the PVM 3 source code, a directory of user supplied examples, and pointers to other PVM related packages are available on *netlib*. Netlib is a software distribution service set up on the Internet There are two 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`.

The second method is to request the PVM software 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 *shar* or uuencoded/compressed *tar* format. In your $HOME area, first type for shar format type

```
% sh pvm3_shar
```

or for tar format type

```
% uudecode pvm3.1.tar.z.uu
% uncompress pvm3.1.tar.Z
% tar xvf pvm3.1.tar
```

This will create a directory called `pvm3`.

3.3. Building

Once you have unpacked the directory called `pvm3` in your $HOME directory, then you are ready to build PVM. The PVM source comes with directories and makefiles for most machines you are likely to have. Generic makefiles are supplied to assist in building PVM for new machines.

Copy the file `pvm3/lib/Cshrc.stub` into your `.cshrc` file on the machines you wish to run PVM. The stub should be placed after your path is defined. This stub automatically determines what type of machine it is on and (optionally) adds the location of PVM to your path.

Building for each architecture type is done automatically by going into the `pvm3` directory and typing `make`. The makefile will automatically determine which architecture it is being executed on and build `pvmd3`, `libpvm3.a`, and `libfpvm3.a`. To build

the dynamic group library `libgpvm3.a`, add 'g' to the default list in the makefile or type `make g`. It places all these files in `pvm3/lib/ARCH`.

To build PVM for the Intel Paragon or iPSC/860 the above instructions work if you are on these machines. Note that node specific versions of the above libraries will also be built `libpvm3pe.a` and libfpvm3pe.a. If you are on a SUN or SGI with Intel cross compilers, then you will need to type `make PGON_x` or `make I860_x`.

3.4. Installing

PVM looks for user executables in the default location `$HOME/pvm3/bin/ARCH`. For example, if a user's PVM application wants to spawn a task called *foo* on a SPARCstation called sunny, then on sunny there should be an executable file `$HOME/pvm3/bin/SUN4/foo`. This default can be changed to a different search path in the hostfile.

If PVM is installed in a single location like `/usr/local` for all users, then each user should still create `$HOME/pvm3/bin/ARCH` to place his own executables in and each user should make a soft link to `pvm3/lib`.

```
% ln -s path_to_pvm3/lib ~/pvm3/lib
```

4. Configuring and Starting PVM

This section contains information necessary for defining an initial configuration of machines to make up the virtual machine. It also contains instructions for starting up PVM and a description of the interactive command monitor, called the PVM console or `pvm` for short.

4.1. Starting PVM

Starting PVM involves executing either `pvm` or `pvmd3`. Without arguments these commands start up pvmd3 on the local host. The console (`pvm`) may be started and stopped multiple times on any of the hosts on which PVM is running, although typically a user starts only one console. The console allows interactive adding and deleting of hosts to the virtual machine as well as interactive starting and killing of PVM processes.

The PVM console prints the prompt

```
pvm>
```

and accepts commands from standard input. The available commands are:

help or **?** 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.

version print version of libpvm being used.

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

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

delete followed by one or more host names deletes these hosts. PVM processes still running on these hosts are lost.

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.

spawn start a PVM application.

kill can be used to terminate any PVM process,

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.

quit exit console leaving daemons and PVM jobs running.

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

The two most popular methods of running PVM 3 are to start `pvm` then add hosts manually or to start `pvmd3` with a hostfile then start `pvm` if desired.

When working from a terminal or single window environment there are several options. The user can run `pvm` and spawn his jobs from within `pvm`. The user can run `pvm` and after defining his configuration quit `pvm`. The user can run `pvmd3` in the background. To start PVM in the background the user types

```
% pvmd3 hostfile &
```

This will not work if `pw` is specified in the hostfile because UNIX will expect input (your password) but UNIX can not handle user input to background jobs.

The following steps get around this problem.

```
% pvmd3 hostfile
```

After the configuration is in place PVM prints `[t80040000] ready`. At this point PVM can be placed in the background by typing control-Z followed by `bg`.

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

4.2. Configuring a Virtual Machine

The `pvmd3` command can take a file name as an argument. This file lists the hosts in the user's virtual machine. In this report we will use the term *hostfile* to mean this file. Optionally, the user can start the PVM console, which in turn will start `pvmd3` on the local machine. From the console hosts can be added manually.

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. The first host listed must be the computer on which you will initially start PVM. 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).

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

Figure 1: Simple hostfile lists virtual machine configuration.

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.

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/ARCH`.

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 all of these options is shown in Figure 2.

4.3. 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.

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

If the message says

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

then the most likely cause is the existance of the file `/tmp/pvmd.<uid>`. This file is used for authentication and should exist only while PVM is running. If pvmd3 is manually killed, then this file is sometimes left behind and prevents PVM from starting. Simply delete this file. Other reasons to get this message include not having PVM installed on a host or problems with your `.rhosts` file. If you do not specify the `pw` option for a particular host, your `.rhosts` file on the remote host must contain the name of the host from which you are starting PVM.

If the message says

```
[t80040000] Login incorrect
```

then it probably means that there is not an account on the remote machine that has the same login name as the local 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, 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
```

4.4. 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.

An example makefile is supplied with the PVM source code in the directory `pvm3/examples`. This makefile demonstrates how C and Fortran PVM applications should be linked with the PVM libraries. The makefile also contains information in its header about additional libraries required on some architectures.

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

4.5. Running PVM Applications

Once PVM is running, an application using PVM routines can be started from a UNIX command prompt on any of the machines in the configuration. An application need

not be started on the same machine the user happens to start PVM.

The standard output and standard error from spawned tasks are written to the log file `/tmp/pvml.<uid>` on the host where PVM was started. If the user wishes to see these writes on his screen then he can execute `tail -f /tmp/pvml.<uid>`. Stdout and stderr appear on the screen for all manually started PVM tasks.

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. 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 3.1 routines can be found on the quick reference guide given in Appendix D. This guide is designed to be removed and folded into a handy reference card.

In this section we give a brief description of the routines in the PVM 3.1 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 the various naming conventions that different Fortran compilers use to call C functions.

Previous PVM versions required that Fortran programmers explicitly terminate all their character strings with NULL if the strings where used as arguments to PVM routines. The Fortran interface takes care of this automatically and is no longer required in PVM 3.

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. The routine pvm_mytid() must be called before any other PVM calls in an application. If PVM has not been started before an application calls pvm_mytid(), then the routine returns an error code.

```
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

0 PvmTaskDefault - PVM chooses where to spawn processes,

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

2 PvmTaskArch - the `where` argument specifies a type of architecture to spawn on using ARCH (see table 1),

4 PvmTaskDebug - starts these processes up under debugger,

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

In Fortran these symbolic names are shortened to **PVMDEFAULT, PVMHOST, PVMARCH, PVMDEBUG, PVMTRACE** and are predefined in parameter statements in the include file **pvm3/include/fpvm3.h**.

PvmTaskTrace is not implemented in PVM 3.1, but it will be available in a later release of PVM 3.x.

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 between the nodes and 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 hostinfo **hostp )
```

```
call pvmfconfig( nhost, narch, 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 `hostinfo` structures. The array is of size `nhost`. Each `hostinfo` structure contains the pvmd tid, host name, name of the architecture, maximum packet length, and relative cpu speed for that host in the configuration.

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

```
call pvmftasks( which, ntask, 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 `taskinfo` structures. The array is of size `ntask`. Each `taskinfo` structure contains the tid, pvmd tid, parent tid, a status flag, and the executable file name.

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. 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 just because `info` was returned negative.

5.4. Signaling

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

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

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

```
int info = pvm_notify( int about, int msgtag, int ntask, int tids )
```

```
call pvmfnotify( about, msgtag, ntask, tids, info )
```

The routine pvm_notify() causes a message with a given `msgtag` to be sent to a set of tasks specified by `ntask` and `tids` when some event happens in the virtual machine. The argument `about` specifies the event type. 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.

5.5. Error Messages

```
int info = pvm_perror( char *msg )
```

```
call pvmfperror( msg, info )
```

This routine prints the error status of the last PVM call. It is similar to the UNIX perror routine.

```
int oldset = pvm_serror( int set )
```

```
call pvmfserror( set, oldset )
```

Calling pvm_serror() with `set` = 1 enables automatic error messages, and any subsequent PVM call that returns an error condition will automatically print the associated error message. Calling pvm_serror() with `set` = 0 disables automatic error detection for subsequent PVM routines. The previous value of the setting is returned in `oldset`. A future option is planned where when `set` = 2 the process prints the error message and then dies.

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.

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 acually receive it.

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.

A new function added in 3.1 is pvm_advise(). This routine "advises" PVM on whether or not to try to use direct task-to-task communication for the subsequent sends. It can be called multiple times during an application to selectively set up direct links, but typical use is to make one call after the initial call to pvm_mytid(). The advantage of direct links is a 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.

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. PVM 2.4 only had one send and receive buffer in its implementation. In PVM 3 there is only 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 used if the virtual machine configuration is determined by PVM to be heterogeneous, otherwise no encoding is done. (same as PVM 2.4 default). In the PVM 3.1 release XDR encoding is always done because of the possibility that another task may be simultaneously adding a computer that will make the configuration heterogeneous.

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. [Not implemented in the initial release] 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.

In Fortran the names used to specify the encodings are shortened to **PVMDEfAULT, PVMRAW, PVMINPLACE** and included as parameter statements in `pvm3/include/fpvm3.h`.

```
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`. 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 pvmfsetrbuf( 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_pkstr(    char   *cp )
```

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, which is of length ntask.

```
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_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_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 )
```

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

specified.

```
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().

```
int info = pvm_advise( route )
```

```
call pvmfadvise( route, info )
```

The routine pvm_advise() advises PVM to set up direct links between tasks. The options for `route` are:

PvmDontRoute - don't allow direct task-task links.

PvmAllowDirect - (default) allow direct links, but don't request them.

PvmRouteDirect - request direct links.

Once a link is established it is used by both tasks and 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. pvm_advise can be called multiple times to selectively establish direct links, but is typically set only once near the beginning of each task.

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_upkstr(    char   *cp )
```

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 these functions. The pvmd does not perform the group functions. This is handled by a group server that is automatically started when a group function is first 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. Explicit group functions are a new feature in PVM 3. Feedback is encouraged with respect to the functionality and flexibility of the supplied routines to meet the group needs of applications. 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 two exceptions are pvm_lvgroup() and pvm_barrier() 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 is 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 continuous 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 continuous 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.

```
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_grgsize() 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. User feedback has requested that pvm_bcast() not send the message back to the sender, which now happens if the sender is a member of the group. In the next release of PVM, pvm_bcast() will not send the message back to the sender.

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.

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 `pvm3/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() to allow it to use the PVM system and enable interprocessor communication. It then calls pvm_spawn() to execute a given number of slave programs on other machines in PVM. Each slave program must also call pvm_mytid() to enable interprocessor communication. Subsequently, pvm_send() and pvm_recv() are used to

pass messages between processes.

When finished, all PVM programs should call pvm_exit() to allow PVM to disconnect any sockets to the processes, 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.

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

```
#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).

machines, a user can get a rough lower bound on the task granularity he should look for in his 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. Functional parallelism is where different machines in a PVM configuration perform 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 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. 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/mandelbrot`. 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.

Two routines, pvm_perror() and pvm_serror() are provided with PVM to allow the automatic detection and printing of any error conditions from PVM routines. Also note that PVM tasks can be started manually under any standard serial debugger, for example dbx.

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, PVM will execute the shell script `pvm3/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 hosts in the virtual machine as specified by the `flag` and `where` arguments in pvm_spawn(). The shell script can be modified to include a preferred debugger or even a parallel debugger if one is available.

Diagnostic print statements sent to either stderr or stdout 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.

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

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 3.1 routines.

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

PVM is composed of a set daemons (pvmd3), which will be referred to as "pvmd", one per host in a user's virtual machine. Each user runs a set of daemons that are configured to work together and so define the collective machine. Pvmds owned by one user do not interact with ones owned by others. PVM was designed for the pvmd to run with a nonprivileged uid and serve a single user to reduce security risk, and also to minimize the impact of one PVM user on another. A virtual machine runs application tasks which are executables that have been linked with libpvm3.a. The pvmd presents a system call-like interface to tasks running on the same host, providing process control, communication, and fault detection.

The overall anatomy of PVM version 3 is shown in Figure 10. `[included separate from the text in pvminternals.ps]` The picture shows a virtual machine composed of three hosts running four tasks, one of which is a "console" that allows the user to do such things as list running tasks or reconfigure the machine. Also shown are the files and communication sockets used, which are detailed later in this chapter.

The pvmds communicate with one another using UDP sockets. A reliable packet-delivery system is implemented on top of UDP. TCP sockets are used between the pvmd and the local PVM tasks, and also directly between tasks on the same host or on different hosts when PVM is "advised" to do so. Native communication routines (using message hardware) are employed inside multiprocessors like the Intel i860. Passing messages using native shared memory will soon be available on machines like the KSR and IBM Power/4.

PVM assumes that each host in a virtual machine is directly connected to each

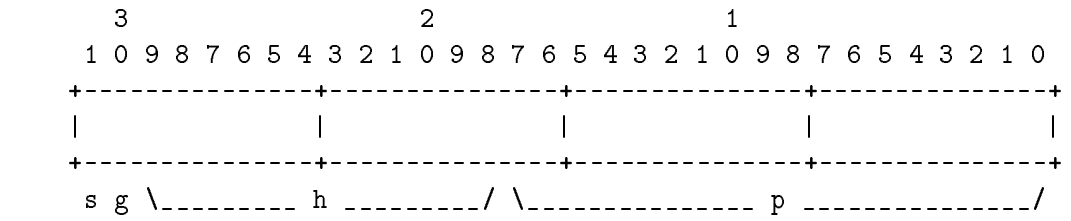
other host via IP, that is, messages are routed directly from any host to any other without going through intermediate pvmds. If necessary, it should be easy to cause a pvmd to route messages through a third party, but this has not been implemented.

10.1. TID Details

PVM uses a 32-bit integer called a task ID (tid) to address pvmds, tasks, and groups of tasks within a virtual machine. A tid uniquely identifies a task within its entire virtual machine, analogous to the way a process id identifies a Unix process on its host. Tids can be recycled but no two tasks will ever have the same tid at the same time.

Tids are intended to be opaque outside the system itself, and the user should not interpret or modify them without using functions supplied in the programming library. The tid is designed to facilitate binding of names to tasks in a distributed fashion and to aid in message routing.

The tid contains four fields as shown below. The diagram is for discussion only - the partitions between fields may someday move (possibly even dynamically) or the fields could appear in a different order. At the time of this writing they are as shown:



Three of the fields, s, g and h, have global meaning, that is, each pvmd of a virtual machine will interpret them the same way. The h field contains a host number relative to the virtual machine. As it joins the group, each pvmd is assigned a unique host number and therefore part of the address space of the machine. The host number zero is used to refer to the localhost or a "shadow" pvmd, used for certain operations, depending on context. The maximum number of hosts in a virtual machine is therefore limited to 2^{h-1} (currently 4095), which is assumed to be enough.

Each pvmd is allowed to assign any local meaning to the p field with the exception that all zeroes is reserved to mean the pvmd itself. On a Unix workstation, for example, the pvmd may put the Unix pid directly into the p field if enough bits are available. On a multiprocessor, the p field could be further divided into a processor field and a partition field. As with the number of hosts, the number of tasks per host is limited by its tid field. Currently, the p field is allotted 18 bits, so at most 262143 tasks can exist concurrently on a host.

The design of the tid allows three things important to the implementation. First, each pvmd can assign tids to local tasks with no off-host communication, as it owns a private part of the address space. Second, a message can be routed directly to its destination host from anywhere in the virtual machine with no address lookup because the host number is encoded in the tid. This helps prevent communication bottlenecks as the size of virtual machines increases, because, for example, no central database of tasks is needed. Finally, pvmds customized to run on multiprocessors can define the

local part of their addresses any way they want, yet their tasks will still be addressable from anywhere in the system.

A tid can hold one of several data, discriminated by ranges of values in the four fields:

	_s_	_g_	_h_	_p_
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 pvmd	1	0	0	0
Multicast address	0	1	1..maxhost	x
Error code	1	1	< small negative number >	

For example, process 3153 on host 12 might have tid 0-0-12-3153.

10.2. Messages

In PVM messages are the building block for all higher-level functions. Approximately half of the code is concerned with packing and unpacking messages, fragmenting them and routing the fragments. Pvmds and tasks use a similar message format, so a task uses the same message packing functions whether composing a message for another task or for its pvmd. A task or pvmd can send a message to any other tid or list of tids, except that communication is not currently allowed between a pvmd and a task not its own, for reasons that will be explained later.

10.3. Libpvm Internals

10.3.1. Connecting to the Pvmd

This section describes how a task connects to its pvmd as implemented in the generic libpvm, i.e. the one used on regular Unix hosts.

First, libpvm reads file `/tmp/pvmd.<uid>`, which is written by the pvmd when it starts up. This contains the sockaddr at which the pvmd will accept connections from tasks, in the form "ip_addr:port". The ip_addr field is normally "7f000001", on the local host. Libpvm connects a socket to this address and sends a message to the pvmd asking to enroll.

When connecting, the pvmd and libpvm must each prove their identity to the other. This is necessary to prevent another user from connecting a task to the pvmd and obtaining service or running a pvmd which masquerades as the real thing. The pvmd and task each create a file owned and writable only by their own uids. They trade filenames and attempt to write in each others' files, then each checks its file. This authentication depends on file permissions and so of course on whether the rogue has

already cracked root or the user's login. Note that it doesn't prevent the user from modifying libpvm so that it "lies"to the pvmd, for example, gives it a bogus process id.

When a task "reconnects" to its pvmd it must supply the pid that the pvmd expects to get, that is, the one returned by fork() during the spawn operation. During the spawn operation, the pvmd creates a context for the task and fills in the pid with that returned by fork(). The context allows messages arriving for the task before it starts up and reconnects to be buffered. When running a task under a debugger, the debugger will generally have the pid returned by fork(), while the task itself will be a child of the debugger, and so will have an unpredictable pid. Before exec()ing a task, the pvmd sets environment variable PVMEPID to the expected pid of the task. If the environment variable PVMEPID is set, libpvm uses the value as its pid for identification to the pvmd, and also sends its real pid.

10.4. Pvmd Internals

10.4.1. Task Management

Each pvmd maintains a list of all tasks running on the host it manages. It keeps a context for each task in the structure below:

```
struct task {
    struct task *t_link;        /* chain */
    struct task *t_rlink;
    int t_tid;                  /* task id */
    int t_ptid;                 /* parent task id */
    int t_flag;                 /* status */
    int t_pid;                  /* unix pid */
    int t_sock;                 /* t-d tcp socket */
    struct sockaddr_in t_sad;   /* addr of sock */
    int t_salen;                /* addr length */
    struct waitc *t_wait;       /* wait channel */
    struct pkt *t_txq;          /* to task */
    struct pkt *t_rxp;          /* pkt reassembly from task */
    struct mesg *t_rxm;         /* to-us msg reassembly from task */
    int t_out;                  /* pipe from stdout/err */
    char *t_authnam;            /* d-auth file name */
    int t_authfd;               /* d-auth file desc */
    char *t_a_out;              /* a.out name (info only) */
    struct mca *t_mca;          /* mca currently in use by task */
}
```

10.4.2. Pvmd Wait Context

Some operations require a pvmd to call another pvmd for service, for example, to check the status of a task on remote host. Since the pvmd cannot block while waiting for the operation to complete, it keeps a record of all relevant data in a "wait context" (waitc)

and continues routing packets and serving other requests. When the reply comes back, it can resume the original operation using the data stashed in the waitc.

The waitc is defined as follows:

```
struct waitc {
    struct waitc *wa_link;      /* chain */
    struct waitc *wa_rlink;
    int wa_wid;                 /* unique identifier */
    int wa_kind;                /* why waiting */
    int wa_on;                  /* entity waiting on */
    int wa_tid;                 /* entity waiting */
    int wa_dep;                 /* remote wait id that depends on us or 0 */
    struct waitc *wa_peer;      /* dll of peers */
    struct waitc *wa_rpeer;
                                /* wait kind-specific stuff below */
    struct mesg *wa_mesg;       /* message being built */
    void *wa_spec;              /* for kind-specific data */
}
```

Each wait context within a pvmd has a unique identifier number, the wait-id (wid). The wid is passed along with a request for service and returned with the reply.

The waitc has some fields that hold data specific to only one or a few operations. The fields common to all operations are `wid`, the serial number of the operation, `kind`, the type of operation, and `tid` and `on`, the tids of the waiter and waitee, respectively.

Wait contexts can be peered together to track operations which involve more than one remote host, for example, to spawn a number of tasks. Each waitc in a peer group privately holds information about a specific remote host, and shares information about the local operation. As each remote pvmd responds, its waitc is freed and the peer group collapses until a single waitc is left. When its pvmd responds, the operation can continue.

10.4.3. Machine Reconfiguration

Hosts can be added to and deleted from the virtual machine at any time, excepting the master host (the initial pvmd), which must be a member for the duration of the machine.

The master pvmd manages the host table, which needs to be synchronized across all the pvmds.

Any task can request a new host be added with the pvm_addhosts() function. The request is relayed to the master pvmd, which attempts to start and configure pvmds on the new hosts and then broadcasts a new host table to all the slave pvmds.

The master pvmd uses either rsh or rexec() to start each new pvmd. Since this operation is subject to hanging and long delays, it spawns a shadow pvmd to do the dirty work. The shadow pvmd is a more-or-less real pvmd, but does not talk to tasks or to any pvmd other than its local peer (the master). It has a special slot (0) in the host table. Once the shadow pvmd does its work, it can send its results back to the

master pvmd via the normal pvmd-pvmd message channel; this avoids having to create another mechanism, such as writing to a file.

As each new pvmd starts up, it trades a small amount of configuration information with the master (shadow) over the connection created by rsh, as follows:

```
master->slave                          slave->master
-------------                          -------------
(rexec) pvmd -s [-dxx]
                                       ddpro<2309>

name<tonka> n<2> ip<80a9c93c:0000> n<1> ip<80a9c9e4:1388> mtu<4096>

                                       arch<SYMM> ip<80a9c93c:1388> mtu<4096>
ep<pvm3/bin/%>
                                                ok
```

Once the slave has a temporary host table, containing only itself and the master, it closes the socket and does the rest using normal messages. It receives an updated host table along with the old slave daemons.

10.4.4. Pvmd Fault Detection

The role of the pvmd in fault detection is to convert a host failure, which might ordinarily cause an application to hang forever, into a detectable event, such as a message. The pvmd itself is not concerned with host failures, since it's essentially a message router and database. The exception is that the master pvmd must be reachable. A few operations require the master pvmd to arbitrate, and no method of handing over the duty of master has been implemented. So, if a pvmd loses contact with the master, it cleans up and exits, knowing that the master will mark it dead.

Failure of a host is detected when trying to send it a message. Acks for packets sent never come back and eventually the pvmd gives up. It edits the failed host out of its host table and completes any operations pending on that host, using information stored in wait context structures. Any further requests for the failed host will return immediately with negative status.

If the machine becomes idle, for example, if all tasks are waiting to receive messages, no messages would be generated to drive fault detection. For this reason, each pvmd has an idle timer and periodically sends packets to other pvmds so that traffic never goes to zero.

Once a host fails, its pvmd is considered dead forever. Another pvmd may be started on the same host at a later time, but it will be considered a different one, that is, there is no way to reclaim any tasks running on the failed host.

10.5. Multiprocessor Interface

PVM version 3 is designed so that native multiprocessor calls can be compiled into the source. This allows the fast message-passing of a particular system to be realized by the PVM application. Messages between two nodes of a multiprocessor should go

directly between them while messages destined for another host on the Internet go to the user's PVM daemon on the multiprocessor for further routing. On shared-memory systems the data movement can be implemented with a shared buffer pool and lock primitives.

We illustrate the use of multiprocessors in PVM version 3 with two examples. In the first example assume a PVM task is running on a node of an Intel Paragon and calls the PVM send routine to send data a PVM task running on another node of the same Paragon. PVM compares the h field of the destination tid with the sender's tid and discovers that the message will stay within the same host. From the p field of the destination tid a native NX message-passing routine is called by PVM to move the message directly to the destination task. In the second example assume the PVM task now sends a message to a task running on a different host out on the Internet. From the h field, libpvm determines the message will not stay within the host so it uses a NX message-passing routine to move the message to the pvmd process running on the service node. The service node of a Paragon has access to the network, and so the pvmd routes the message to the PVM daemon running on the destination host. The remote daemon then delivers the message to the destination task using the p field of the destination tid. In this way a PVM application can have tasks scattered across several heterogeneous parallel computers and still send messages between all the tasks as though they were all on one large parallel computer.

The PVM daemon that runs on a multiprocessor listens for messages coming from both the outside network and the nodes of the multiprocessor. The efficiency with which this can be implemented depends on the specific operating system and message-passing software. Getting the pvmd to talk to tasks on the nodes is likely the biggest hurdle in porting PVM to a multiprocessor. For example, vendors' communication routines are often not compatible with Unix file I/O using select(). One obvious solution is to put the pvmd in a tight loop polling both interfaces. This works fine when a lot of communication is going on, but eats up just as much cpu time when the tasks are idle.

The pvmd might instead send directly to tasks on nodes but receive through a buffer process. The buffer process would block waiting for messages from the nodes, and forward them to the pvmd, which could then wait in its normal select() loop. This imposes the penalty of an extra copy, however.

A hybrid solution could allow the pvmd to poll during busy times, and so transfer messages directly without any extra copies, but "power down" when the tasks are not talking to it. The pvmd would wait in its normal select() loop, timing out only every 300 msec to poll the nodes for messages. When a message arrives, the pvmd would switch to polling and receiving as fast as it can route packets, until no more messages are ready. Then it switches back to occasional polling. This effectively increases the latency in communicating with an MPP node from outside, but does not degrade the bandwidth, at a small cost in cpu time. The polling could possibly be eliminated by having a second process, not in the direct data path, signal the pvmd when data is ready from a node.

The essential steps of multiprocessor integration are:

1. Define the local meaning of the tid p field. A pvmd only has to understand its own local tid part, which might contain information such as partition and

processor number. On some systems, identifiers will fit directly into the tid p field, otherwise some sort of mapping will be necessary.

2. Modify the pvmd and libpvm so they can send messages to one another using the native message passing.

3. Modify the pvmd so it can create processes, which may require it to allocate a partition, load node processors, etc. Define the semantics of pvm_spawn on the MPP.

4. Modify the libpvm so it can route messages to another libpvm running on the same MPP without going through the pvmd.

Higher-level operations should port more easily as they attempt to stay above the message transport system. It should be possible to incrementally modify the system, first to allow it to run and then to optimize performance.

10.6. Instrumentation

There are several unsupported debugging features built into the pvmd and libpvm. These are only briefly documented and may change at any time. However, it may be useful to have short descriptions.

Malloc wrappers

To facilitate development, the pvmd and libpvm use a sanity-checking malloc library, `imalloc`. The imalloc functions are wrappers for the regular libc functions malloc, realloc, and free. Upon detecting any of several errors, the imalloc functions abort the program so the fault can be found with a debugger.

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 spurious free()s of the same space.

3. I_malloc() and i_realloc() write pads with a pseudo-random pattern at the head and tail of each block, which are checked by i_free() to attempt to discover 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.

6. The heap space can be dumped and optionally sanity-checked at any time by calling i_dump(). This aids tracing memory leaks.

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

Debug masks

The pvmd and libpvm are capable of generating debugging logs. Each contains a mask that may be set to enable various print statements; bits in the mask enable classes of events. The pvmd has an option, -d, which sets the debug mask. The debug mask value of the master pvmd is inherited by slave pvmds. The debugmask in libpvm can be set by calling pvm_setdebug().

Pvmd data structures

The console has a hidden command, "tickle", that pokes the pvmd to dump certain data structures. Typing `help tk` at the console prompt will produce a list of available pokes.

11. Support

Several avenues exist for getting help with using PVM. A PVM bulletin board exists 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, KSR, and Thinking Machines have decided to supply and support PVM software on their systems. The PVM developers also supply PVM support 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 from netlib@ornl.gov in the pvm3 directory.

12. Appendix A. Reference pages for PVM 3.0 routines

This appendix contains an alphabetical listing of all the PVM 3.0 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. Pvmd must already be installed and the user must have an account on the specified machines.

`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 zero indicate an error.

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 0. If some error occurs then `info` will be < 0. 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 his 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 his 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.

Both pvm_addhosts and pvm_delhosts are expensive operations that require the synchronization of the entire virtual machine. Not only do all the other computers have to be aware of a change in the configuration, but any operations presently being done by a computer about to be deleted must be handled in a graceful manner.

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.
PvmSysErr	local pvmd is not responding.
PvmOutOfRes	PVM has run out of system resources.

pvmfadvise() pvm_advise()

advises PVM to use direct task-to-task routing or not.

Synopsis

C `int info = pvm_advise( int route )`
Fortran `call pvmfadvise( route, info )`

Parameters

`route` – integer advising PVM to set up direct task-to-task links.

`route` options		
`PvmDontRoute`	1	Don't allow direct links to this task
`PvmAllowDirect`	2	Allow but don't request direct links
`PvmRouteDirect`	3	Request direct links

`info` – integer returning error status.

Discussion

The routine `pvm_advise` advises PVM on whether or not to set up direct task-to-task links (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. pvm_advise 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 advise 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. pvm_advise returns the error status in `info`.

The performance of direct task-to-task links can be up to a factor of two better than the default route. The draw back is a lack of scalability of the direct links. Some versions of UNIX limit the number of links to no more than 30.

Examples

C:

```
info = pvm_advise( PvmRouteDirect );
```

Fortran:

```
CALL PVMFADVISE( PVMROUTEDIRECT, INFO )
```

Errors

This error condition can be returned by `pvm_advise`

Name	Possible cause
PvmBadParam	giving an invalid route value.

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.

`pvm_bcast` immediately 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`. If the calling task is a member of the group the message will also be sent to itself in version 3.0 and 3.1. In the next PVM release the broadcast messaeg will not be sent 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.

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 actual message label. Useful when the message was received with a wildcard msgtag.
`tid`	–	integer returning the source of the message. Useful when the message was received with a wildcard tid.
`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

This error condition can be returned by `pvm_bufinfo`.

Name	Possible cause
PvmNoBuf	There is no active receive buffer.

pvmfconfig() pvm_config()

returns information about the present virtual machine configuration.

Synopsis

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

Fortran
```
call pvmfconfig( nhost, narch, 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 which contain information about each host including its pvmd task ID, name, architecture, maximum packet length, and relative speed.

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

Examples

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

Fortran:
```
CALL PVMFCONFIG( NHOST,NARCH,INFO )
```

Errors

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 zero indicate an error.

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 before the computer is deleted. If pvm_delhosts is successful, `info` will be 0. If some error occurs then `info` will be < 0. 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 deleting this host from the virtual machine. The status of hosts can be requested by the application using pvm_mstat and pvm_config. It is still the responsibility of the application developer to make his application tolerant of host failure.

Both pvm_addhosts and pvm_delhosts are expensive operations that require the synchronization of the entire virtual machine. Not only do all the other computers have to be aware of a change in the configuration, but any operations presently being done by a computer about to be deleted must be handled in a graceful manner.

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; otherwise, PVM will not know if the process has crashed or exited normally.

Examples

C:

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

Fortran:

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

Errors

Name	Possible cause
PvmSysErr	pvmd not responding

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 and tables 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 should 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 arrive 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.

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.

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`

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`

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.

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 3.0, 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.

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 if heterogeneous
`PvmDataRaw`	1	no encoding
`PvmDataInPlace`	2	data left in place

Option names are shortened in Fortran to:

Encoding value		MEANING
`PVMDEFAULT`	0	XDR if heterogeneous
`PVMRAW`	1	no encoding
`PVMINPLACE`	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`. With the default setting if all the computers in the virtual machine use the same data format then no encoding is performed; otherwise, XDR encoding is used when packing the message. The other options to encoding allows 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.

In the initial release of PVM 3.0 the default option always does XDR conversion because PVM can not know if the user is going to add a heterogeneous machine

before this message is sent. The PvmDataInPlace is also not implemented in the initial release.

These and other options may be available as future extensions to PVM. If a user or vendor have their own optimized encoding routines and wish certain messages to use these routines, then these calls can be added to the PVM source under a new encoding option. For example, Cray Research has fast vectorized encoding routines between Cray 64 bit and IEEE 32 bit formats that could exploit this option.

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 PVM3.0 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.

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 (SIGKILL) 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.

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.

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. 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_mcast` multicasts a message stored in the active send buffer to `ntask` tasks specified in the `tids` array. If the sending task's tid is in the array, then the message will also be sent to itself. 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 and based on a minimum spanning tree algorithm between the pvmds. 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 a spanning tree broadcast is performed 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 if heterogeneous
`PvmDataRaw`	1	no encoding
`PvmDataInPlace`	2	data left in place

Option names are shortened in Fortran to:

Encoding value		MEANING
`PVMDEFAULT`	0	XDR if heterogeneous
`PVMRAW`	1	no encoding
`PVMINPLACE`	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 if all the computers in the virtual machine use the same data format then no encoding is performed; otherwise, XDR encoding is used when packing the message. 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.

In the initial release of PVM 3.0 the default option always does XDR conversion because PVM can not know if the user is going to add a heterogeneous machine before this message is sent. The PvmDataInPlace is also not implemented in the initial release.

Other options may be available as future extensions to PVM. If a user or vendor have their own optimized encoding routines and wish certain messages to use these routines, then these calls can be added to the PVM source under a new encoding option. For example, Cray Research has fast vectorized encoding routines between Cray 64 bit and IEEE 32 bit formats that could exploit this option.

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

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).

enrolls this process into PVM on its first call and returns the *tid* of the process on every call.

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 `pvm_mytid` must be the first PVM call in any program that uses PVM 3.0 routines. The routine enrolls this process into PVM on its first call. It also 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.

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()

notify a set of tasks about some event.

Synopsis

C
```
int info = pvm\_notify( int what, int msgtag,
                        int ntask, int *tids )
```

Fortran
```
call pvmfnotify( what, msgtag, ntask, 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.

`ntask` – integer specifying the length of the tids array.

`tids` – integer array of length `ntask` that contains a list of task or pvmd tids to be notified.

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

Discussion

The routine `pvm_notify` is a general notification function. After pvm_notify is called, if the specified event occurs, then a message is generated and multicast to all tasks listed in the `tids` array. The tasks are responsible for receiving the message with the specified tag `msgtag` and taking some appropriate action. Future versions of PVM may expand the list of available notification events, and allow user defined events.

Examples

C:
```
info = pvm_notify( PvmHostAdd, 9999, ntask, tids )
```

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

Errors

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

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. 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 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.

Examples

C:

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

Fortran:

```
      CALL PVMFNRECV( -1, 4, ARRIVED )
      IF ( ARRIVED .NE. 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.

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

Synopsis

C

```
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_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

`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.

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.

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.

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.

In SPMD (hostless) programs pvm_parent can be used to determine if a given program instance needs to spawn copies of itself. Examples in the Users' Guide and source code illustrate this use.

In a master/slave paradigm pvm_parent can be used by the slaves to determine the tid of the master so they can send results back.

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.

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`.

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. 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 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 )
  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.

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`.

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.

blocks until a message with specified message tag has arrived from the specified source, and places it in a new active receive buffer.

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.

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`

immediately 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. 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_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.

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.

pvmfserror() pvm_serror()

sets automatic error message printing on or off for subsequent PVM calls by this process.

Synopsis

C `int oldset = pvm_serror( int set )`
Fortran `call pvmfserror( set, oldset )`

Parameters

`set` – integer defining whether detection is to be turned on (1) or off (0).
`oldset` – integer defining the previous setting of pvm_serror.

Discussion

The routine `pvm_serror` sets automatic error message printing for all subsequent PVM calls by this process. Any PVM routines that return an error condition will automatically print the associated error message. The argument `set` defines whether this detection is to be turned on (1) or turned off (0) for subsequent calls. In the future a value of (2) will cause the program to exit after printing the error message. pvm_serror returns the previous value of set in `oldset`.

All stdout and stderr messages are placed in the file /tmp/pvml.<uid> on the master pvmd's host.

Examples

C:

```
info = pvm_serror( 1 );
```

Fortran:

```
CALL PVMFSERROR( 0, INFO )
```

Errors

This error condition can be returned by `pvm_serror`

Name	Possible cause
PvmBadParam	giving an invalid set value.

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.

switches the active send buffer.

Synopsis

C `int oldbuf = pvm_setsbuf( int bufid )`
Fortran `call pvmfsetsbuf( bufid, oldbuf )`

Parameters

`bufid` – integer the 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/architecture_name/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. *

In Fortran `flag` should be the **sum** of:

Option value		MEANING
`PVMDEFAULT`	0	PVM can choose any machine to start task
`PVMHOST`	1	`where` specifies a particular host
`PVMARCH`	2	`where` specifies a type of architecture
`PVMDEBUG`	4	start up processes under debugger
`PVMTRACE`	8	processes will generate PVM trace data. *

* future extension

`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 postive 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`. 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.

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 will use the metrics of machine load and rated performance to determine the most appropriate hosts.

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 is `~/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 taskinfo **taskp )
struct taskinfo{
    int  ti_tid;
    int  ti_ptid;
    int  ti_host;
    int  ti_flag;
    char *ti_a.out;
} taskp;
```

Fortran

```
call pvmftasks( where, ntask, info )
```

Parameters

`where` – integer specifying what tasks to return information about. The options are:

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, and the name of this task's executable file. The status flag values are: waiting for a message, waiting for the pvmd, and running.

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

Discussion

The routine `pvm_tasks` returns information about tasks presently running on the virtual machine. The information returned is the same as that available from the console command `ps`.

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

Examples

C:

```
info = pvm_tasks( 0, &ntask, &taskp );
```

Fortran:

```
CALL PVMFTASKS( DTID, NTASK, INFO )
```

Errors

The following error condition 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.

unpack the active message buffer into arrays of prescribed data type.

Synopsis

C

```
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

`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.

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.

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.

13. Appendix B. Mapping of PVM 2.4 routines to PVM 3.0 routines

Appendix B

This appendix contains a mapping of PVM 2.4 routines to the equivalent PVM 3 routines. It is intended to assist programmers in converting their existing PVM applications to the latest release and thus gain the advantage of improved performance and portability.

In the case where there is not a direct mapping, a code fragment performing an equivalent function in PVM 3 is giving.

Both C and Fortran mappings are given.

Mapping PVM 2.4 to PVM 3 in C

PVM 2.4	PVM 3
Process Control	
`inum = enroll( task )`	`tid = pvm_mytid()`
`inum = initiate( task, arch )`	`numt = pvm_spawn( task,0,2,arch,1,&tid )`
`inum = initiateM( task, host )`	`numt = pvm_spawn( task,0,1,host,1,&tid )`
`info = whoami( task, inum )`	`tid = pvm_mytid()`
`info = pstatus( &ncpu, &narch )`	`info = pvm_config( &ncpu, &narch, (struct hostinfo**)0 )`
`status = status( task, inum )`	`status = pvm_pstat( tid )`
`info = terminate( task, inum )`	`info = pvm_kill( tid )`
`leave()`	`pvm_exit()`
Synchronization	
`info = barrier( name, cnt )`	`inum = pvm_joingroup( name )`
	`info = pvm_barrier( name, cnt )`
`info = waituntil( sname )`	`info = pvm_recv( ready_tid, ready_tag )`
`info = ready( sname )`	`info = pvm_mcast( waiting_tids, ntid, ready_tag )`
Sending	
`initsend()`	`info = pvm_initsend( PvmDataDefault )`
`info = putbytes(  xp, cnt )`	`info = pvm_pkbyte(  xp, cnt, 1 )`
`info = putncplx(  cp, cnt )`	`info = pvm_pkcplx(  cp, cnt, 1 )`
`info = putndcplx(  zp, cnt )`	`info = pvm_pkdcplx(  zp, cnt, 1 )`
`info = putndfloat( rp, cnt )`	`info = pvm_pkdouble( rp, cnt, 1 )`
`info = putnfloat(  rp, cnt )`	`info = pvm_pkfloat(  rp, cnt, 1 )`
`info = putnint(  ip, cnt )`	`info = pvm_pkint(  ip, cnt, 1 )`
`info = putnlong(  ip, cnt )`	`info = pvm_pklong(  ip, cnt, 1 )`
`info = putnshort(  jp, cnt )`	`info = pvm_pkshort(  jp, cnt, 1 )`
`info = putstring(  sp )`	`info = pvm_pkstr(  sp )`
`info = snd( task, inum, msgtag )`	`info = pvm_send( tid, msgtag )`
`info = snd( task, -1, msgtag )`	`info = pvm_mcast( tid_array, ntid, msgtag )`
Receiving	
`mt = probe( msgtag )`	`bufid = pvm_probe( -1, msgtag )`
`mt = rcv( msgtag )`	`bufid = pvm_recv( -1, msgtag )`
`info = rcvinfo( &bytes, &msgtag, task, &inum )`	`info = pvm_bufinfo( bufid, &bytes, &msgtag, &tid )`
`info = getbytes(  xp, cnt )`	`info = pvm_upkbyte(  xp, cnt, 1 )`
`info = getncplx(  cp, cnt )`	`info = pvm_upkcplx(  cp, cnt, 1 )`
`info = getndcplx(  zp, cnt )`	`info = pvm_upkdcplx(  zp, cnt, 1 )`
`info = getndfloat( rp, cnt )`	`info = pvm_upkdouble( rp, cnt, 1 )`
`info = getnfloat(  rp, cnt )`	`info = pvm_upkfloat(  rp, cnt, 1 )`
`info = getnint(  ip, cnt )`	`info = pvm_upkint(  ip, cnt, 1 )`
`info = getnlong(  ip, cnt )`	`info = pvm_upklong(  ip, cnt, 1 )`
`info = getnshort(  jp, cnt )`	`info = pvm_upkshort(  jp, cnt, 1 )`
`info = getstring(  sp )`	`info = pvm_upkstr(  sp )`

Mapping PVM 2.4 to PVM 3 in Fortran

PVM 2.4	PVM 3
Process Control	
`call fenroll( task, inum )`	`call pvmfmytid( tid )`
`call finitiate( task, arch, inum )`	`call pvmfspawn( task,2,arch,1,tid,numt )`
`call finitiateM( task, host, inum )`	`call pvmfspawn( task,1,host,1,tid,numt )`
`call fwhoami( task, inum )`	`call pvmfmytid( tid )`
`call fpstatus( ncpu, narch, info )`	`call pvmfconfig( ncpu, narch, info )`
`call fstatus( task, inum, status )`	`call pvmfpstat( tid, status )`
`call fterminate( task, inum, info )`	`call pvmfkill( tid, info )`
`call fleave()`	`call pvmfexit( info )`
Synchronization	
`call fbarrier( name, cnt, info )`	`call pvmfjoingroup( name, inum )`
	`call pvmfbarrier( name, cnt, info )`
`call fwaituntil( sname, info )`	`call pvmfrecv( ready_tid, ready_tag, info )`
`call fready( sname, info )`	`call pvmfmcast( ntid, waiting_tids, ready_tag, info )`
Sending	
`call finitsend()`	`call pvmfinitsend( PVMDEFAULT, info )`
`call fputbytes( xp, cnt, info )`	`call pvmfpack( BYTE1, xp, cnt, 1, info )`
`call fputncplx( cp, cnt, info )`	`call pvmfpack( COMPLEX8, cp, cnt, 1, info )`
`call fputndcplx( zp, cnt, info )`	`call pvmfpack( COMPLEX16, zp, cnt, 1, info )`
`call fputndfloat( rp, cnt, info )`	`call pvmfpack( REAL8, rp, cnt, 1, info )`
`call fputnfloat( rp, cnt, info )`	`call pvmfpack( REAL4, rp, cnt, 1, info )`
`call fputnint( ip, cnt, info )`	`call pvmfpack( INTEGER4, ip, cnt, 1, info )`
`call fputnlong( ip, cnt, info )`	`call pvmfpack( INTEGER4, ip, cnt, 1, info )`
`call fputnshort( jp, cnt, info )`	`call pvmfpack( INTEGER2, jp, cnt, 1, info )`
`call fputstringl( sp, cnt, info )`	`call pvmfpack( STRING, sp, cnt, 1, info )`
`call fsnd( task, inum, msgtag, info )`	`call pvmfsend( tid, msgtag, info )`
`call fsnd( task, -1, msgtag, info )`	`call pvmfmcast( ntid, tid_array, msgtag, info )`
Receiving	
`call fprobe( msgtag, mt )`	`call pvmfprobe( -1, msgtag, bufid )`
`call frcv( msgtag, mt )`	`call pvmfrecv( -1, msgtag, bufid )`
`call frcvinfo( bytes, msgtag, task, inum, info )`	`call pvmfbufinfo( bufid, bytes msgtag, tid, info )`
`call getbytes( xp, cnt, info )`	`call pvmfunpack( BYTE1, xp, cnt, 1, info )`
`call getncplx( cp, cnt, info )`	`call pvmfunpack( COMPLEX8, cp, cnt, 1, info )`
`call getndcplx( zp, cnt, info )`	`call pvmfunpack( COMPLEX16, zp, cnt, 1, info )`
`call getndfloat( rp, cnt, info )`	`call pvmfunpack( REAL8, rp, cnt, 1, info )`
`call getnfloat( rp, cnt, info )`	`call pvmfunpack( REAL4, rp, cnt, 1, info )`
`call getnint( ip, cnt, info )`	`call pvmfunpack( INTEGER4, ip, cnt, 1, info )`
`call getnlong( ip, cnt, info )`	`call pvmfunpack( INTEGER4, ip, cnt, 1, info )`
`call getnshort( jp, cnt, info )`	`call pvmfunpack( INTEGER2, jp, cnt, 1, info )`
`call getstringl( sp, cnt, info )`	`call pvmfunpack( STRING, sp, cnt, 1, info )`

14. Appendix C. PVM 3.0 Quick Installation Guide

This appendix contains a quick installation guide for PVM 3. A more detailed discussion and installation troubleshooting are include in Sections 3 and 4 of the Users' Guide. In the following description `ARCH` represents the PVM architecture name, for example SUN4, (see table 1 for a complete list of ARCH names).
The following 4 steps should be performed for each architecture class that will be used with PVM 3. [Note the use of NFS allows steps 1 and 2 and 4 to be skipped for cross mounted machines.]

Step 1. unpack pvm3 directory in your HOME directory

Step 2. insert the file pvm3/lib/Cshrc.stub into your .cshrc file after your path variable has been defined. This stub automatically sets `ARCH` and (optionally) adds the correct path to `pvm` and `pvmd3`

Step 3. cd to pvm3 and type `make`. This will build and install `pvmd3`, `pvm`, `libpvm3.a`, `libfpvm3.a`, and `libgpvm3.a` for this architecture. (Some versions require the user to type 'make g' to build the group library)

Step 4. install `pvm` and `pvmd3` files on all computers of this architecture class. Note this is not necessary for machines that are NFS together. The default location is `$HOME/pvm3/lib/ARCH`. This default can be changed in the last line of the Cshrc.stub file. Insert the stub file into your .cshrc file on each computer.

15. Appendix D. PVM 3.0 Quick Reference Guide