Message Passing Toolkit: PVM Programmer's Manual

Document Number 007–3686–002

New Features

Message Passing Toolkit: PVM Programmer's Manual 007–3686–002

This rewrite of the *Message Passing Toolkit: PVM Programmer's Manual*, supports the 1.3 release of the Cray Message Passing Toolkit and the Message Passing Toolkit for IRIX (MPT). The MPT implementation of PVM for UNICOS, UNICOS/mk, and IRIX systems contained in this release is based on the Oak Ridge National Laboratories (ORNL) version 3.3.10.

With MPT release 1.3, support for XPVM has been dropped.

Record of Revision

Version	*Description*
1.0	January 1996 Original Printing.
1.1	August 1996 This revision supports the Message Passing Toolkit (MPT) 1.1 release.
1.2	January 1998 This revision supports the Message Passing Toolkit (MPT) 1.2 release for UNICOS, UNICOS/mk, and IRIX systems.
1.3	February 1999 This revision supports the Message Passing Toolkit (MPT) 1.3 release for UNICOS, UNICOS/mk, and IRIX systems.

Contents

Page

Tables

About This Manual

This publication documents the Cray Message Passing Toolkit and Message Passing Toolkit for IRIX (MPT) 1.3 implementation of PVM-3 supported on the following platforms:

- Cray PVP systems running UNICOS release 10.0 or later. The MPT 1.3 release requires a bugfix package to be installed on UNICOS systems running release 10.0 or later. The bugfix package, `MPT12_OS_FIXES`, is available through the `getfix` utility. It is also available from the anonymous FTP site `ftp.cray.com` in directory `/pub/mpt/fixes/MPT12_OS_FIXES`.
- CRAY T3E systems running UNICOS/mk release 1.5 or later.
- Silicon Graphics MIPS based systems running IRIX release 6.2 or later. IRIX 6.2 systems running PVM require the POSIX patch set and any patches recommended by the patch set.

This implementation of PVM-3 is based on the public domain PVM product, version 3.3.10, developed by researchers at the Oak Ridge National Laboratory (ORNL), the University of Tennessee (UT), and Emory University (EU). It consists of a PVM library and several commands that support PVM.

Related Publications

The following documents contain additional information that might be helpful:

- *Message Passing Toolkit: MPI Programmer's Manual*
- *NQE User's Guide*
- *NQE Administration*
- *Application Programmer's Library Reference Manual*
- *Installing Programming Environment Products*

All of these are Cray publications and can be ordered from Cray. For ordering information, see "Ordering Publications."

Other Sources

Material about PVM is available from the following other sources:

- *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*, available at the following URL:

  `http://www.netlib.org/pvm3/book/pvm-book.html`

- Usenet news group at `comp.parallel.pvm`
- PVM standard, available from the Computer Science and Mathematics Division of Oak Ridge National Laboratories.
- PVM related web pages from the following PVM home page:

  `http://www.epm.ornl.gov/pvm`

Ordering Publications

The *User Publications Catalog* describes the availability and content of all Cray hardware and software documents that are available to customers. Customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, call +1 651 683 5907. Silicon Graphics employees may send electronic mail to `orderdsk@sgi.com` (UNIX system users).

Customers who subscribe to the CRInform program can order software release packages electronically by using the `Order Cray Software` option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers:

1	User commands
1B	User commands ported from BSD
2	System calls
3	Library routines, macros, and opdefs
4	Devices (special files)
4P	Protocols
5	File formats
7	Miscellaneous topics
7D	DWB-related information
8	Administrator commands

Some internal routines (for example, the `_assign_asgcmd_info`() routine) do not have man pages associated with them.

variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

In this manual, references to Cray PVP systems include the following machines:

- CRAY C90 series
- CRAY C90D series
- CRAY EL series (including CRAY Y-MP EL systems)
- CRAY J90 series
- CRAY Y-MP E series
- CRAY Y-MP M90 series
- CRAY T90 series

Silicon Graphics systems include all MIPS based systems running IRIX 6.2 or later.

The following operating system terms are used throughout this document.

Term	Definition
UNICOS	Operating system for all configurations of Cray PVP systems
UNICOS/mk	Operating system for all configurations of CRAY T3E systems
UNICOS MAX	Operating system for all configurations of CRAY T3D systems
IRIX	Operating system for all configurations of MIPS based systems

The default shell in the UNICOS and UNICOS/mk operating systems, referred to in Cray Research documentation as the *standard shell*, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2–1992
- X/Open Portability Guide, Issue 4 (XPG4)

The UNICOS and UNICOS/mk operating systems also support the optional use of the C shell.

Cray UNICOS version 10.0 is an X/Open Base 95 branded product.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and part number of the document with your comments.

You can contact us in any of the following ways:

- Send electronic mail to the following address:

  `techpubs@sgi.com`

- Send a facsimile to the attention of “Technical Publications” at fax number +1 650 932 0801.

- Use the Suggestion Box form on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com/library/`

- Call the Technical Publications Group, through the Technical Assistance Center, using one of the following numbers:

  For Silicon Graphics IRIX based operating systems: 1 800 800 4SGI

  For UNICOS or UNICOS/mk based operating systems or CRAY Origin2000 systems: 1 800 950 2729 (toll free from the United States and Canada) or +1 651 683 5600

- Send mail to the following address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043–1389

We value your comments and will respond to them promptly.

Overview [1]

The Cray Message Passing Toolkit and Message Passing Toolkit for IRIX (MPT) is a software package that supports interprocess data exchange for applications that use concurrent, cooperating processes on a single host or on multiple hosts. Data exchange is done through *message passing*, which is the use of library calls to request data delivery from one process to another or between groups of processes.

The MPT 1.3 package contains the following components and the appropriate accompanying documentation:

- Parallel Virtual Machine (PVM)
- Message Passing Interface (MPI)
- Logically shared, distributed memory (SHMEM) data-passing routines

The Parallel Virtual Machine (PVM) software was initially developed to enable a collection of heterogeneous computer systems to be used as a coherent and flexible concurrent computation resource. Silicon Graphics has taken this initial implementation and extended it in several ways.

This chapter provides an overview of the PVM software that is included in the toolkit.

1.1 The PVM Package

This manual contains instructions for building, installing, and using the MPT implementation of PVM-3. The MPT version of PVM supports IRIX, UNICOS, and UNICOS/mk systems. It consists of a PVM library and several commands that support PVM. The most important of these is a user-level daemon that runs on each computer system in the PVM system.

The MPT version of PVM contains architecture-specific enhancements that target Cray PVP systems. The PVM library can also function as a stand-alone library within a single Cray PVP machine. This stand-alone library makes use of Cray multitasking software, offering enhanced communication performance by allowing PVM tasks to communicate through memory instead of through sockets.

For IRIX systems, the MPT version of PVM has enhancements to use POSIX shared memory, which provides greater flexibility and robustness than did the previously used IRIX shared arenas.

By default, for UNICOS and UNICOS/mk systems, communication is based on data transfers over UNIX domain sockets between UNIX processes on the same system or over TCP sockets between processes on different systems. For IRIX systems, the default communication is based on TCP sockets between processes on the same system and between different systems. Transfer speeds are relatively slow when sockets are used as the mechanism for communication. The MPT version of PVM also provides alternative mechanisms for communication, such as memory for communication within a UNICOS system. The socket communication has been optimized to utilize high-speed network devices more effectively. The different communication mechanisms are discussed further in the PVM man pages, and the communication costs (in time, resources, and so on) associated with the different communication mechanisms are discussed in Chapter 2, page 5.

PVM has been integrated with the Network Queuing Environment (NQE) so that you can use PVM within a batch job in isolation from other PVM jobs. On UNICOS systems you can use NQE load balancing for choosing the virtual machine and for placing spawned tasks. This is described in more detail in Chapter 2, page 5. For more information about NQE, see the *NQE User's Guide*, and *NQE Administration*.

On UNICOS/mk systems, the PVM library can also function as a stand-alone library within a single executable file. This mode allows you to use PVM to communicate among PEs within a multiple PE process (that is, a single executable file). This mode uses `shmem` calls to communicate between PEs.

1.2 PVM on Silicon Graphics Systems

As described in this manual, Silicon Graphics provides versions of PVM to support a variety of needs. These versions provide users with a single subroutine interface for message passing programming; this interface is portable and a de facto standard. PVM is available from its developers as public domain software and is being made available as vendor-supported software by Silicon Graphics and a number of other computer vendors. By using PVM in your application, you can avoid being locked into a proprietary interface.

PVM is supported on all Silicon Graphics systems. The PVM software system consists of a library and commands that support PVM. The PVM software provided by Silicon Graphics has been developed specifically for each system on which it runs.

1.3 PVM Terminology and Scenarios

You may choose to use PVM to communicate among processes on a number of different computer systems. With PVM available on all types of Silicon Graphics systems and a large number of systems from other vendors, you have a large number of combinations of systems and clusters of systems available.

However, in the context of Silicon Graphics systems, this large number of combinations can be grouped into five basic scenarios. Each scenario describes a particular combination of systems that can be defined by the form and cost of the underlying communications mechanism used by PVM.

The following terminology is used in describing these scenarios and elsewhere in this manual:

Term	Definition
task	The entity that uses PVM for communications. This entity can be a UNIX process or a Cray multitasked task.
application	A number of tasks running the same program.
process	The entity running on the UNICOS, UNICOS/mk, or IRIX operating system or another UNIX system.

The following scenarios are listed approximately in order of increasing cost of communication:

1. Two processes running on a single UNICOS system. (PVM can use either networking capabilities or memory for communication.)

2. One executable file running on a UNICOS/mk or UNICOS MAX system. (PVM uses high-performance connections between the tasks on the processing elements in the partition.) This scenario requires only the CRAY T3D version of PVM, as part of the Programming Environment, or Cray MPT version for UNICOS/mk systems in stand-alone mode.

3. One or more executable files running on an IRIX system. (POSIX shared memory is used between processes.) This scenario requires MPT for IRIX.

4. One process running on a UNICOS, IRIX, or UNICOS/mk system and a second process running on a second UNICOS, IRIX, or UNICOS/mk system. (PVM is used across the network between the two systems.)

5. Two processes on separate partitions on a single UNICOS/mk system. PVM uses the network to communicate between the two partitions. This scenario requires the MPT version of PVM.

6. One process running on a UNICOS system and another process running on an associated UNICOS MAX system. (PVM connects across the channel between the two systems.) This scenario requires the MPT version and the CRAY T3D version of PVM, available from the Programming Environment.

7. Two processes running on separate partitions on a single UNICOS MAX system. (PVM uses network communications through an associated UNICOS system.) This scenario requires the MPT version and the CRAY T3D version of PVM.

The preceding scenarios represent a range of simple uses of PVM; more complex scenarios involving combinations of those described can easily be formed. The following characteristics apply to all PVM scenarios:

- The user building an executable file for use on a Silicon Graphics system links with a single PVM library, regardless of how PVM is used.

- The same standard library syntax and behavior are supported, regardless of how PVM is used (although certain releases may support features not appropriate to other releases).

- The performance of PVM in different basic scenarios differs significantly; this difference influences the communications strategy that should be used.

PVM Functionality [2]

This chapter describes the Message Passing Toolkit (MPT) implementation of the Parallel Virtual Machine (PVM) software. The following concepts are discussed:

- Multiple computer systems as a virtual machine
- Applications and environments
- PVM program development
- Data types
- Environment variables

2.1 Multiple Computer Systems As a Virtual Machine

PVM is a software system that enables a collection of heterogeneous computer systems to be used as a coherent and flexible concurrent computation resource. The individual systems can be shared-memory or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations interconnected by a variety of networks. From the user's point of view, the combination of these different systems can be treated as a single *virtual machine* when using PVM. The term *host* refers to one of the member computer systems.

PVM support software executes on each system in a user-configurable pool and presents a unified, general, and powerful computational environment for concurrent applications. User programs, written in C or Fortran programming languages, gain access to PVM in the form of library routines for functions such as the following:

- Process or task initiation
- Message transmission and reception
- Synchronization through the use of barriers or rendezvous

Optionally, users can control the execution location of specific application components; the PVM system transparently handles message routing, data conversion for incompatible architectures, and other tasks that are necessary for operation in a heterogeneous, networked environment.

2.2 Applications and Environments

PVM is ideally suited for concurrent applications composed of many interrelated subalgorithms, although performance is good even for traditional parallel applications. PVM is particularly effective for heterogeneous applications that exploit specific strengths of individual systems on a network. As a loosely coupled, concurrent supercomputing environment, PVM is a viable scientific computing platform.

PVM has been used for molecular dynamics simulations, superconductivity studies, distributed fractal computations, matrix algorithms, and as the basis for teaching concurrent programming.

2.3 PVM Program Development

To develop a program that uses PVM, you must perform the following steps:

1. Add PVM function calls to your application for process initiation, communications, and synchronization. For syntax descriptions of these functions, see Chapter 4, page 63.
2. Build executable files for the systems that you will use, as described in Section 2.3.1, page 7.
3. Create a host file to define the virtual machine, as described in Section 2.3.2, page 7.
4. If your program is in distributed mode, execute the PVM daemon and your application in one of the following ways:
   - As described in Section 2.3.4, page 12, for the PVM daemon, and as described in Section 2.3.5, page 13, for your application
   - As an NQS job, as described in Section 2.3.6, page 14
   - Through the PVM console by using the console `spawn` command, as described in Table 2, page 17
5. If your application is in stand-alone mode, execute it as described in Section 2.3.9.2, page 30, or Chapter 3, page 43.
6. Troubleshoot the application, if necessary. For information on PVM troubleshooting, see Section 2.3.8, page 19.
7. Optimize the application as described in Section 2.3.9, page 23.

2.3.1 Building PVM Executable Files

After you have added PVM function calls, a simple UNICOS PVM code can be linked as follows:

```
cc -o compute -lpvm3 compute.o
```

This command links the `compute.o` object code with the PVM library and creates an executable file named `compute`.

For IRIX systems, if you begin with the source file, you must specify the `-I` (include) option and the Application Binary Interface (ABI) of the application development library (N32 or 64 ABIs), as follows:

```
cc -I /usr/array/PVM/include -64 -o compute compute.c -lpvm3
```

For IRIX systems, if you begin with an object file, the code can be linked as follows:

```
cc -64 -o compute compute.o -lpvm3
```

If you have the optional IRIX `mpt` module loaded, use the following command:

```
cc -64 -o compute compute.c -lpvm3
```

After the code is linked, you can install the executable files on the Silicon Graphics systems you will be using. If you specified the `ep` option in the host file for a system, install the file in the specified directory. Otherwise, install it in the following directory:

```
$HOME/pvm3/bin/$PVM_ARCH
```

2.3.2 Creating Host Files

Each system in the PVM virtual machine must have a separate entry in the host file. Lines that begin with a hash symbol (#), possibly preceded by white space, are ignored.

If you do not want PVM to start a host immediately, but you might start it later by using the `pvm_addhosts`(3) function or the PVM console `add` command, you do not need to include the host in the host file. However, if you need to set any of the options described in Table 1, page 9, you should include the specified system in the host file, preceded by the ampersand (&) character.

A simple host file can be created automatically on UNICOS systems if NQE load balancing is available. Start the PVM daemon with the following command:

```
pvmd3 -h &
```

This command starts the PVM daemon in the background and tells it that automatic host file selection should be used. On UNICOS systems, the PVM daemon queries the load balancing server for available hosts and includes all available hosts in your virtual machine. Available hosts are determined by the PVM policy in NQE or by a policy specified in the `PVM_POLICY` environment variable. Hosts can be excluded based on many different resources. For more information on NQE policies, see *NQE Administration*. If a host file is also specified, PVM uses the options specified in the host file. A host specified in the host file will be included in the virtual machine only if that host is available, as determined by the NQE policy.

Example 1, is an example of a host file that contains the names of systems, which is the basic information necessary in a host file.

Example 1: Simple Host File

```
# my first host file
thud
fred
wilma
gust.cray.com
rain
```

You should verify that no system is listed more than once, and that the system on which the master `pvmd3`(1) daemon will run (the *master host*) is included in the host file (see Section 2.3.4, page 12, for information on starting the `pvmd3` daemon). Automatic host file selection always includes the host running the master `pvmd3`(1) daemon.

The `$PVM_ROOT` and `$PVM_ARCH` environment variables are set for you automatically when you load the `mpt` module to access the Message Passing Toolkit software. To customize your environment, you can specify the options listed in Table 1, after any system name in the host file.

Table 1. Host File Options

Option	Description
`bx=` *dpath*	Specifies the debugger path. You can also set this path by using the `PVM_DEBUGGER` environment variable. The default debugger path is `$PVM_ROOT/lib/debugger`.
`dx=` *loc*	Specifies a location for `pvmd3` other than the default, `$PVM_ROOT/lib/$PVM_ARCH/pvmd3`. This option is useful in debugging new versions of PVM. The *loc* variable may be a simple file name, an absolute path name, or a path relative to the user's home directory on the remote system. The `pvmd3` daemon is installed in `$PVM_ROOT/lib/$PVM_ARCH/pvmd3` when the MPT version is installed on Silicon Graphics systems.
`ep=` *paths*	Specifies a series of paths to search for application tasks. A percent sign (`%`) in the path expands to the architecture of the remote system. Multiple paths are separated by a colon (`:`). By default, PVM looks for application tasks in the following directories: `$HOME/pvm3/bin/$PVM_ARCH:$PVM_ROOT/bin/$PVM_ARCH`
`ip=` *network_name*	Specifies the network name to be used for communication. The default is determined by the network routing, as shown by the `netstat -i` command. You can use this option to specify HIPPI or another specific device.
`lo=` *userid*	Specifies an alternative login name for the system. The default is the login name on the master system.
`so=ms`	Causes the master `pvmd3` daemon to request that you manually start a `pvmd3` daemon on a slave system when the `rsh`(1) and `rexec`(1) network services are disabled but IP connectivity exists. The default is no request. You cannot start the master system from the PVM console or background when you specify this option. (This option is rarely used.)
`so=pw`	Causes PVM to prompt for a password on the remote system. This option is useful when you have a different login name and password on a remote system. The master host prompts you for your password, as in the following example: `Password(honk.cs.utk.edu:manchek):` Type your password for the remote system. The startup will then continue as normal. You cannot start the master host from the PVM console or background when you specify this option.

Option	Description
`sp=` *value*	Specifies the relative computational speed of this system compared to other systems in the configuration. *value* is an integer in the range 1 through 1,000,000. The default is 1000. (This option currently has no effect on PVM operation.)
`wd=` *path*	Specifies the path name of a working directory in which all spawned tasks on the host will execute. The default is `$HOME`.

A dollar sign (`$`) in an option introduces an environment variable name, for example, `$PVM_ARCH`. Each PVM daemon expands names from environment variables.

The simple host file in Example 1, page 8, works well if both of the following conditions are met:

- You have a login with the same name on all of the systems in your host file.
- The local system is listed in the `.rhosts` file on each of the remote systems.

To supply an alternative login name for the `thud` system, add the `lo` option to its host file entry, as follows:

`thud lo=`*NAME*

To be queried for your password on a system named `cyclone`, add `so=pw` to its host file entry, as follows:

```
cyclone so=pw
```

To specify the path of the daemon executable file for a system named `sun114`, add the `dx` option, as follows:

```
sun114 dx=/usr/fred/pvm3/lib/Sun/pvmd3
```

Note: By default, the MPT version of `pvmd3` is installed in `$PVM_ROOT/lib/$PVM_ARCH/pvmd3` on Silicon Graphics systems, where `$PVM_ROOT` and `$PVM_ARCH` are set for you automatically when you load the `mpt` module.

The string specified in the previous example is passed to a shell so that variable expansion works. Following is another example that uses variable expansion:

```
sun114 dx=bin/$MYBIN/pvmd3
```

You can change the default value of any option for all hosts in a host file by specifying them on a line with an asterisk (*) in the host field, as in the following example:

```
thud.cs.utk.edu
gust.cray.com
sun114 dx=/tmp/pvmd3
* lo=afriend so=pw
```

The preceding example sets the default login name (on remote systems) to `afriend` and queries for a password on each system. Defaults set in this way are effective forward from the location at which they occur in the host file. They can be changed with another * line.

You can override the location of executable files by adding the `ep` option to your host file entries, as in the following example:

```
ep=$HOME/pvm3/bin
```

Unlike the `dx` option, which names the daemon file, the `ep` option names a directory.

Example 2 shows a more complex host file in which host names are followed by options.

Example 2: Sample Host File with Host Name Options

```
# host file for testing on various platforms
# default to my executable
*                  dx=pvm/SUN4/pvmd3
fonebone
refuge
sigi.cs            dx=pvm/PMAX/pvmd3
# reset default for other systems
*                  dx=$PVM_ROOT/lib/$PVM_ARCH/pvmd3
# do not start this system, but define ep in case we add it later
& rain.cray.com  ep=$(HOME)/bin ip=rain-hippi
# borrowed accts, "guest", don't trust fonebone
*                  lo=guest  so=pw
sn666.jrandom.com  ep=$(HOME)/bin
cubie.misc.edu     ep=pvm/IPSC/pvmd3
```

2.3.3 Specifying Architecture Types

Before you run a PVM executable file on an IRIX system, you must specify the architecture type by setting the `PVM_ARCH` environment variable. Four architecture types are supported for IRIX systems. With the software installed in the default locations, you must also set the `PVM_ROOT` environment variable to `/usr/array/PVM` and the `PATH` environment variable to `$PVM_ROOT/lib/$PVM_ARCH`. The following C shell example shows the setting of all three variables:

```
setenv PVM_ARCH SGIMP64
setenv PVM_ROOT /usr/array/PVM
setenv PATH ${PATH}:${PVM_ROOT}/lib/$PVM_ARCH
```

The architecture types shown in the following list are arranged in an approximate order of lowest to highest performance types:

Architecture type	Description
`SGI32`	N32 ABI/MIPS III version using sockets
`SGI32mips4`	N32 ABI/MIPS IV version using sockets
`SGIMP64mips3`	64 ABI/MIPS III version using POSIX shared memory and sockets
`SGIMP64`	64 ABI/MIPS IV version using POSIX shared memory and sockets

2.3.4 Starting and Stopping the PVM Daemon

After you have written a host file, you can start up the master `pvmd3`(1) daemon by passing it the host file as an argument. You must specify the appropriate path for `pvmd3`(1). On a Silicon Graphics system, for example, you can enter one of the following:

`pvmd3` *hostfile* `&`

or

`pvm [`*hostfile*`]`

If you do not specify a host file when starting the PVM console, the PVM daemon found in the default location will be started on the local machine.

The ampersand (&) in the first line tells the operating system to run `pvmd3`(1) in the background, which is what you will normally want to do.

You should not run `pvmd3`(1) in the background if you have to enter passwords for any of the slave systems (that is, if you included the `so=pw` option for one or more systems). In this case, run `pvmd3`(1) in the foreground and then stop it (by pressing `CONTROL-Z`) and put it in the background (by entering `bg` at the prompt) after all systems have started up.

To shut off PVM, enter `halt` at a PVM console prompt. For detailed information on using console prompts, see Section 2.3.7.2, page 16.

If the master `pvmd3`(1) daemon has trouble starting a slave `pvmd3`(1) daemon on a system, the error message written to the PVM log file from the master `pvmd3`(1) may indicate the problem.

2.3.5 Running PVM Applications

When the `pvmd3`(1) daemon is running successfully, you can start your application. PVM provides the following methods of starting applications:

- Start the application from the shell command line.

  With this method, you start the application as any command or application would be started. For example, if the application is named `a.out`, enter the following command at the shell command line prompt:

  ```
  ./a.out
  ```

- Start the application from the PVM console by using the `spawn` command.

  With this method, you first start the console. After the `pvm>` prompt has appeared, enter the `spawn` command followed by the application name or path, as needed. For example, to run an application named `cannon`, enter the following command at the console command line prompt:

  ```
  spawn cannon
  ```

You can obtain help for the `spawn` command by typing `help spawn` at the console command line prompt.

Once the application has started, it displays standard output and standard error information for the initial task, but not for the other tasks in the application. PVM captures this output information and sends it to the master daemon. The daemon, in turn, prefaces each line with a PVM task identifier that identifies its source, and writes it to the PVM log file.

The log file can contain very useful information about the virtual machine and its tasks. By default, the log file contains output from the PVM daemon, including error messages and output from tasks. Optionally, the log file can contain debugging output from the daemon.

When PVM is run without NQS, the log file is located in `/tmp`. When PVM is run without NQE on UNICOS and UNICOS/mk systems, the log file is located in `/tmp/pvml.`*uid*, where *uid* is the user ID. The IRIX implementation allows overlapping PVM virtual machines. Therefore, more than one PVM daemon started by the same user can run on the same host. The log file is located in `/tmp/pvml.`*uid.vmid*, where *uid* is the user ID and *vmid* is the virtual machine ID. By default, is 0, but if the `PVM_VMID` (formerly `PVMJID`) environment variable is set, *vmid* will equal the numeric value of `PVM_VMID`.

When PVM is run using NQS on UNICOS and UNICOS/mk systems, the name of the log file is `$TMPDIR/pvml.`*uid*. `$TMPDIR` is a temporary directory created for the NQS job. You can examine this file at any time, but remember that each task buffers the output written to standard output independently (unless you flush the output after each write request), and so the sequence of output from two different tasks may vary.

Instead of having the data written to the PVM log file, you can request that output be sent as a PVM message to another task's output device. For more information, see the `PvmOutputTid` and `PvmOutputCode` options on the `pvm_setopt`(3) man page.

You can also redirect output by using options on the console `spawn` command (see Table 2, page 17) or by using the `pvm_catchout`(3) function. For more information about running your program in stand-alone mode, see Section 2.3.9.2, page 30, or Chapter 3, page 43.

2.3.6 Using NQS to Run PVM Applications

On UNICOS, UNICOS/mk, and IRIX systems, PVM applications can be run as part of an NQS job script. Each NQS job has its own PVM daemon; therefore, the PVM daemon must be started within the NQS job script. This is different from interactive use, in which one daemon is run per user per system. Any application run as part of the same NQS job script uses the same PVM daemon. Slave daemons that run on UNICOS and UNICOS/mk systems will also run one daemon per NQS job. IRIX users can achieve similar functionality as UNICOS and UNICOS/mk users by using the `PVM_VMID` environment variable. Using `PVM_VMID` allows more than one daemon to run per user per system.

In UNICOS and UNICOS/mk implementations, a special environment variable is checked for a batch (NQE) environment. If a batch environment exists, these implementations place the PVM log and daemon (`pvmd` socket) files in a special temporary directory. In IRIX implementations, a single user running multiple NQE jobs on a single host should set the `PVM_VMID` environment variable for each batch job.

PVM processes spawned by the daemon inherit the limits of the NQS job. This allows a user to run multiple NQS jobs that use PVM, each with limits of the NQS job being run. Previous versions of PVM used the same daemon for multiple NQS jobs.

The following example is an NQS job script to run the application `foo`:

```
module load mpt
pvmd3 hostfile &   # Start the daemon
sleep 60           # Wait for startup
foo                # Run application
pvm << EOF         # Start console to halt pvm
halt
EOF
```

2.3.7 Using the PVM Console

Using the PVM console is an alternative to using the `pvmd3`(1) command to start the daemon and execute your application. The `pvm`(1) command starts the console, which can be started and stopped multiple times on any of the systems on which PVM is running.

2.3.7.1 Starting the Console

Start the PVM console by using the following command line:

`pvm [`*hostfile*`]`

When the console is started, it checks to see if a PVM daemon is running. If so, it simply attaches itself to the daemon and can be used to monitor ongoing PVM processes as shown:

```
% pvm
pvmd already running
pvm>
```

If the daemon is not started, the `pvm`(1) command tries to start one, but the command must first find the daemon. (Currently, the `pvm`(1) command does not examine the *hostfile* argument, if provided, but simply passes its name to the daemon. Therefore, the pvm command cannot use information from this file.)

The logic used by the pvm command to start the daemon is as follows:

1. The command tries to execute $HOME/pvm3/lib/pvmd on all systems. $HOME/pvm3/lib/pvmd must be an executable file that is one of the following:

   - A shell script that starts up the PVM daemon, perhaps by using a host file. If you use this option, you may find it useful to have the script do other preparatory or related work.
   - A symbolic link to the PVM daemon. The following example shows how you can set up a link on Silicon Graphics systems:

   ```
   % mkdir ~/pvm3
   % mkdir ~/pvm3/lib
   % ln -s $PVM_ROOT/lib/$PVM_ARCH/pvmd3 ~/pvm3/lib/pvmd
   ```

2. On Silicon Graphics systems, if pvmd3(1) is not found or cannot be executed, the pvm(1) command explicitly tries to start $PVM_ROOT/lib/$PVM_ARCH/pvmd3. This special processing is not performed on other systems.

   a. If a daemon is started, you see the following:

   ```
   % pvm
   pvm>
   ```

   b. If a daemon is not started, you see the following:

   ```
   % pvm
   libpvm [pid-1]: Console: Can't start pvmd
   %
   ```

2.3.7.2 Using Console Commands

When you enter the pvm(1) command, the console responds with a prompt and accepts the commands described in Table 2.

Table 2. Console Commands

Command	Description	
`add` *hostnames*	Adds systems to the virtual machine.	
`alias[`*name command* `[`*args*`]]`	Defines or lists console command aliases.	
`conf`	Lists the PVM system configuration. Fields in the output from `conf` are as follows:	
	`HOST`	Host name
	`DTID`	PVM daemon task identifier
	`ARCH`	PVM system name (architecture)
	`SPEED`	Relative speed of this system
`delete` *hostnames*	Deletes systems from the virtual machine. PVM processes that are still running on these systems are lost.	
`echo [` *args* `]`	Echoes arguments.	
`halt`	Kills all PVM processes and shuts down PVM; all daemons exit. This is the best way to exit the console if you are done using PVM. See `quit`.	
`help [` *command* `]`	Provides minimal information about the console commands. If you enter `help` followed by a command name, a brief description of the syntax is displayed.	
`id`	Prints the `pvm_tid` task identifier of the console. (The console is simply another PVM task.)	
`jobs [-l]`	Displays a list of running jobs. The `-l` option provides more detailed output.	
`kill [-c ]`*taskids*	Kills a PVM user process. The `-c` option indicates that children of the task IDs should also be killed.	
`mstat` *hostnames*	Gives status for each system listed.	
`ps [-a] [-h` *host*`] [-n` *host*`] [-l][-x]`	Gives a listing of current processes and their status. The following options are available:	
	`-a`	All systems (default is local)
	`-h` *host*	Task ID of the system (with no blanks)
	`-n` *host*	System name (with no blanks)

Command	Description		
		This example illustrates `-n` *host* usage:	
		`ps -ngust`	
		This command requests the status of a system named `gust.`	
	`-l`	Shows long output	
	`-x`	Shows console task	
	`ps` output includes the following fields:		
	`HOST`	System executing the process	
	`A.OUT`	Executable name (if known to PVM)	
	`TID`	Task identifier	
	`PTID`	Parent's task identifier (`-l` only)	
	`PID`	Task process identifier (`-l` only)	
	`FLAG`	Process status. Can be one or more of the following:	
		`a`	Task is waiting for authorization.
		`c`	Task is connected to `pvmd`.
		`o`	Task connection is being closed.
		`H`	Host starter task is identified.
		`R`	Resource manager task is identified.
		`T`	Task starter task is identified.
`pstat` *tid*	Displays the status of the specified PVM process.		
`quit` (or EOF)	Exits the console, but leaves the daemons and processes running. See `halt`.		
`reset`	Resets the virtual machine. Causes a `SIGKILL` signal to be sent to every running process. All message queues are cleared. The `pvmd` daemons are left in an idle state.		
`setenv [`*name* `[` *value* `]]`	Displays or sets environment variables.		
`sig` *num task*	Sends a signal to specified tasks.		

Command	Description
`spawn` [*options*]*file*	Starts a PVM application for the specified file. Options are as follows: `-` *count* — Number of tasks (default is 1) `-` *host* — Spawn on *host* `-` *arch* — Spawn on hosts of *arch* `-?` — Enables debugging `->` — Redirects output of job to console `->` *file* — Redirects output of job to *file* `->>` *file* — Appends output of job to *file* If NQE load balancing is available, the `spawn` command places tasks based on the load balancer, but within the restrictions specified on the `spawn` command. In the following example, the `spawn` command spawns four instances of `a.out` on the system named `gust`. `pvm>` **`spawn -4 -gust a.out`**
`trace` [*names*]	Sets or displays a trace event mask. The *names* argument refers to names defined in the PVM include file, `$PVM_ROOT/include/pvmtev.h.` Alternatives are as follows: `trace [+]` *names* `trace [-]` *names* `trace [+] *` `trace [+] *`
`unalias` *name*	Undefines the specified command alias.
`version`	Displays the `libpvm` version.

2.3.8 Troubleshooting PVM

This section describes common problems encountered when using PVM and provides suggested solutions. There are several kinds of problems that can keep `pvmd3`(1) from building a virtual machine. The most common are permission problems.

If you do not specify the `pw` option for a particular system, your `.rhosts` file on that system must contain the name of the host from which you start the master `pvmd`. Otherwise, you will get a message like one of the following (although you may not get the entire message):

```
pvmd3@hostname: Permission denied
```

```
pvmd3@hostname: Login incorrect
```

To get the entire error message, enter the following command at a shell prompt:

```
rsh hostname daemon
```

On UNICOS and UNICOS/mk systems, you need to use `remsh`(1) rather than `rsh`(1) because the name `rsh` is used for a restricted shell, not for the remote shell command. The `remsh` command is not available on IRIX systems. *daemon* is the location of the PVM daemon (for example, `/tmp/pvm/pvmd3` or `$PVM_ROOT/lib/$PVM_ARCH/pvmd3`).

Look at the output of the command and consult whichever of the following sections most closely applies.

2.3.8.1 PVM Already Running

When you start the `pvmd3`(1) daemon, you may receive a message that PVM is already running because a file exists in `/tmp`. If no `pvmd3`(1) is running, it is likely that the last time you used PVM you did not terminate `pvmd3`(1) by using the console `halt` command, or the previous execution of the `pvmd3` daemon terminated abnormally, leaving the files in `/tmp`. Remove the file named in the message and start `pvmd3`(1) again.

NQS jobs on UNICOS or UNICOS/mk systems place this file in the `$TMPDIR` directory, which is automatically deleted at the end of the job. Slave daemons of NQS jobs on these systems also use `$TMPDIR`, which is set by the login process. IRIX systems use `/tmp`. Messages about slave daemon startup failures are placed in the PVM log file.

2.3.8.2 `pvmd3` Fails to Start on Remote System

If you use a shell (such as `.kshrc`) that does not automatically execute a startup script that sets `$PVM_ROOT` on added hosts, you can set the `PVM_DPATH` environment variable to the full or relative path of the `pvmd` startup script, or include the `dx` option in the host file to specify the path to the startup script. The `pvmd` startup script automatically sets `$PVM_ROOT` on the remote host.

The following command shows how to set the `PVM_DPATH` environment variable:

```
setenv PVM_DPATH $PVM_ROOT/lib/pvmd
```

The following command shows how to specify the `pvmd` startup script in the host file:

```
dx=/opt/ctl/mpt/mpt/pvm3/lib/pvmd
```

Note: The `dx` option in the host file overrides the `PVM_DPATH` environment variable, and `$PVM_ROOT` is not acknowledged for `dx`, so the `dx` path must be a full pathname.

2.3.8.3 Permission Denied

If you get a message denying you permission, it probably means that your `.rhosts` file on the remote system does not include your local system name. Add a line like the following to your `.rhosts` file on the remote system:

local-host-name your-local-user-name

Sometimes a system has more than one name, and the remote system may think your local system has a name that is different from the one that you have specified. To determine the name of your local system on the remote system, execute `telnet`(1) or `rlogin`(1) to get to the remote system and enter the following UNIX command:

```
% who am i
```

Look at the last column of the output of this command, which contains the first 16 characters of what the remote system (the one to which you connected) thinks is the name of your local system (the one on which you entered `telnet`(1) or `rlogin`(1)). Make sure you put that system name (the full name, not just the first 16 characters) in your `.rhosts` file on the remote system. Your `/etc/hosts` file should contain the full name. If you do not have this file, see your system administrator for the name. Some older systems require that you spell the name exactly the same, including the case; newer systems accept the name in either uppercase or lowercase.

2.3.8.4 Login Incorrect

If you get a message saying your login is incorrect, there is probably no account on the remote system that has the same login name as your login name on the local system. In this case, you need to add a `lo=` *username* option to your PVM host file.

2.3.8.5 Version Incorrect

If you get a message about a version mismatch, it indicates that the versions of PVM on the two systems were built from different PVM releases. You may be building with an old library, accessing an old PVM version built from the public domain version, or having some similar problem. Ensure that the versions of PVM on the two systems are compatible.

As a general rule, releases of the public domain implementation of PVM with the same second digit in the version number (for example, 3.2.0 and 3.2.6) will interoperate. Changes that result in incompatibility are held until a major version change (for example, from version 3.2 to version 3.3). For compatibility, you might need to upgrade one of your versions of PVM.

2.3.8.6 Failure of Spawn

A common application problem is the failure of a `pvm_spawn ()` request. The PVM console command `tickle 6 4` enables tracing of spawn requests. The complete executable path is printed in the PVM log file.

2.3.8.7 Other Problems

If you get any other messages, ensure that your `.cshrc` file on the remote system is not printing something out when you log in or is not trying to set your terminal characteristics (usually by using the `stty`(1) or `tset`(1) commands).

If you want to print from your `.cshrc` file when you log in, put the relevant commands in an `if` statement in your `.cshrc` file, as in the following example:

```
if ( { tty -s } && $?prompt )  then
# example of printing something when you log in
 echo terminal type is $TERM
# example of setting terminal attributes
   stty erase '^?' kill '^u' intr '^c' echo endif
```

This statement ensures that printing occurs only when you log in from a terminal (and when you are not running a `csh`(1) command script).

2.3.9 Optimizing Use of PVM

Several PVM functions are particularly useful when developing applications that involve UNICOS and UNICOS/mk systems. This section discusses some techniques that can help improve performance. As is true with programming in general, optimization with PVM involves trade-offs. Generally, the trade-off involves reducing generality in favor of better performance. Adding some of the optimizations discussed in the following sections will improve performance but will make the application harder to move to different PVM virtual machines or other systems.

2.3.9.1 Running PVM on UNICOS Multiprocessor Systems

When multiple PVM processes run on a UNICOS multiprocessor system, the processes use sockets, by default, to communicate. The UNICOS operating system recognizes that the transfer is local and uses a faster path, but the overhead is still quite significant. If the PVM processes are executing different executable files, this is probably your only choice.

In some applications, the PVM processes each execute the same executable file, with the work sent out by a master process. If these processes are not communicating with each other (perhaps they are communicating only with the master), this kind of general approach may work well. But if the processes need to communicate with one another, your overall performance will decrease when sockets are used for communication.

A shared memory implementation of the MPT version of PVM is offered to provide better performance for applications in which PVM processes need to communicate with each other. The shared memory implementation of PVM uses macrotasking so that communication between spawned processes can be done through memory instead of sockets. This implementation is available on UNICOS systems. Memory provides a mechanism for communicating between PVM processes on UNICOS systems that is faster than other mechanisms that involve the operating system. Because current UNICOS systems do not have hardware for System V shared memory support, this implementation of PVM uses the Cray multitasking software to imitate a shared memory system.

Two modes of execution are available with the shared memory implementation of PVM. In addition to the standard mode of operation, you can run the shared memory implementation of PVM in a stand-alone mode of operation that requires no PVM daemon or console. This mode provides the best performance for applications that consist of a single executable file and that execute within a single UNICOS machine. Stand-alone mode closely resembles the current PVM mode of operation on the UNICOS/mk system. Because PVM task management from outside the application itself is unnecessary, you can run an application by simply typing `a.out`. If an application follows the master/slave model (consisting of multiple executable files), it might be desirable to convert the application to run within a single executable file to get the best performance.

Modules were used to install the toolkit on your system. To access the shared memory implementation of PVM, the `mpt` module must be loaded.

To modify a PVM program to make use of shared memory, perform the following steps:

1. Convert all global and static data to `TASKCOMMON` data.

   In the public domain version of PVM, all data is assumed to be private to each PVM task. Communication between tasks is done by sending messages. However, in a multitasking environment, all members of the multitasking group can access all global or static data because they share one user address space.

   To preserve the behavior of the public domain version of PVM as much as possible, all global or static data that can be modified during the course of execution of a program must be treated as data local to each task. This is done by placing the data in `TASKCOMMON` blocks. *TASKCOMMON storage* is a mechanism that is used in multitasked programs to provide a separate copy of data for each member of the multitasking group. `TASKCOMMON` data is still globally accessible across functions within a multitasked task, but it is private to that task.

   Fortran examples of global or static data that must be placed in `TASKCOMMON` storage are data that resides in `COMMON` blocks and data that appears in `DATA` or `SAVE` statements. In C, you must place all data that is declared static (either locally or globally) or data declared at a global level (outside of any function) in `TASKCOMMON`.

   Because changing your program so that all global and static data is private is both tedious and makes a program less portable, you can use compile-time command line options to do the conversions. Most global and static data can be converted automatically to `TASKCOMMON` data by using the following command-line options:

   - For C programs:

     ```
     cc -h taskprivate
     ```

- For Fortran programs:

  ```
  f90 -a taskcommon
  ```

Note: Software included in the 1.0 release of the Message Passing Toolkit is designed to be used with the Cray Programming Environment. When building an application that uses the shared memory version of PVM, you must be using the Programming Environment 3.0 release or later. Before you can access the Programming Environment, the `PrgEnv` module must be loaded. For more information on using modules, see *Installing Programming Environment Products*, or, if the Programming Environment has already been installed on your system, see the online ASCII file `/opt/ctl/doc/README`.

When you are placing data in `TASKCOMMON` storage, there may be cases in which the compiler cannot do the conversion because of insufficient information. The compiler notes these cases by issuing a warning during compilation. For such cases, you must convert the data by hand. Most of the time, these cases are related to initialization that involves Fortran `DATA` or `SAVE` statements or C initialized static variables, and you might need to change only how or when the data is initialized for it to be placed in `TASKCOMMON`.

The following is an example of a case that the compiler cannot handle:

```
int a;
int b = &a
```

If variable `a` resides in `TASKCOMMON`, its address will not be known until run time; therefore, the compiling system cannot initialize it. In this case, the initialization must be handled within the user program.

2. Use one of the following methods to request shared-memory PVM process initiation.

   - Stand-alone mode:

     Add a call to the `start_pes ()` function at the beginning of the PVM program.

     This function is provided as a general process initiation function that can be used to start processes for shared memory (SHMEM) data-passing applications and PVM message passing applications that run in the stand-alone mode of operation. The `start_pes` function starts tasks the first time it is called and is not operational on subsequent calls. `start_pes ()` has one argument, *npes*. This argument specifies the total number of tasks with which to run the program. If *npes* is 0, the function starts a number of tasks indicated by an environment variable called `NPES`. This environment variable allows more flexibility because the number of PEs to use on the application can be changed at run time.

   - Standard mode:

     Add a `PvmMtSpawn` flag to a call to the `pvm_spawn ()` function.

     This flag specifies that spawned PVM processes are to be started in a new multitasked group. This is convenient in master-slave applications in which one master starts multiple slaves, and the slaves are set up for fast communication because they are threads in a multitasked group instead of separate user processes. The spawning process uses the `fork`(2) and `exec`(2) system calls to spawn the slave executable file in the master/slave model, and then uses the macrotasking `TSKSTART`(3F) routine to spawn further slave processes, creating a multitasking group for the slave executable file.

3. Use the `cc`(1) or `f90`(1) commands to build your shared memory PVM program, as in the following examples:

   C programs:

```
cc -htaskprivate -D_MULTIP_ -L$MPTDIR/lib/multi -I$PVM_ROOT/include file.c
```

   For C programs, the `-D` and `-L` options are needed to access the reentrant version of `libc` that is required to provide safe access to `libc` routines in a multitasking environment. When the `mpt` module is loaded, the module software sets `$MPTDIR` automatically and points to the default MPT software library. To make compiling in C easier, the environment variable `$LIBCM` is also set automatically when the `mpt` module is loaded. You can use $LIBCM with the cc(1) command to request the reentrant version of libc. $LIBCM is set to the following value:

```
-D_MULTIP_ -L$MPTDIR/lib/multi
```

The following example uses $LIBCM:

```
cc -htaskprivate $LIBCM -I$PVM_ROOT/include file.c
```

Fortran programs:

```
f90 -ataskcommon -I $PVM_ROOT/include file.f
```

4. Select private I/O if private Fortran file unit numbers are desired.

   Note: Automatic TASKCOMMON conversion and private I/O are available in the Programming Environment release 3.0 or later.

   In a multitasking environment, Fortran unit numbers are, by default, shared by all members of the multitasking group. This behavior forces all files to be shared among PVM tasks that were spawned using multitasking. Allowing PVM tasks to share files can be useful, but this behavior is different from that of the public domain version of PVM. The user can request that files be private to each PVM task by specifying the private I/O option on the assign(1) command. The examples in Table 3, page 28, request private I/O.

Table 3. `assign` Examples

Example	Description
`assign -P private u:10`	Specifies that unit 10 should be private to any PVM task that opens it.
`assign -P private p:%`	Specifies that all named Fortran units should be private to any PVM task that opens them. This includes all units connected to regular files and excludes units such as 5 and 6, which are connected to `stdin`, `stdout`, or `stderr` by default.

`assign -P global u:0`
`assign -P global u:5`
`assign -P global u:6`
`assign -P global u:100`
`assign -P global u:101`
`assign -P global u:102`
	This set of `assign` commands can be used in conjunction with `assign -P private g:all` to retain units connected by default to `stdin`, `stdout`, and `stderr` as global units. A unit connected to these standard files cannot be a private unit.

For more information on private I/O functionality on UNICOS systems, see the `assign`(1) man page.

5. Use one of the following methods to run the application:

   • Stand-alone mode PVM applications

     To run an application that uses the shared memory version of PVM in the stand-alone mode of operation, simply type `a.out`. If you have included a call to `start_pes ()` with 0 as the number of PEs to initiate, the `NPES` environment variable must be set before execution. Because PVM applications that run using the stand-alone mode of operation are of fixed size and composition, the support of some PVM functions is not appropriate.

     The following functions are not supported in stand-alone mode:

     `pvm_addhosts ()`

     `pvm_catchout ()`

     `pvm_delhosts ()`

     `pvm_getfds ()`

`pvm_hostsync ()`

`pvm_kill ()`

`pvm_mstat ()`

`pvm_notify ()`

`pvm_reg_hoster ()`

`pvm_reg_tasker ()`

`pvm_sendsig ()`

`pvm_spawn ()`

`pvm_tidtohost ()`

These functions are permitted in programs but return a `PvmNotImpl` status.

- Standard mode (master/slave) PVM applications:

  A master/slave application that uses the shared memory implementation of PVM is run as it is with the public domain version of PVM.

  The master task and the PVM daemon are not multitasked and will communicate with the multitasked slave PVM tasks by means of sockets. The slave PVM tasks that were spawned by the master program are multitasked and can communicate with each other through memory. By default, all PVM tasks communicate with the daemon and other nonmultitasked PVM tasks by using sockets. Note, however, that UNICOS limits the number of open files per application and the number of open sockets in the system. Socket communication is very slow, especially compared to the speed of communication between multitasked PVM tasks. Because much of socket communication is single-threaded, the performance cost goes up as more PVM tasks try to communicate at the same time.

  For these reasons, it might be desirable to change a program so that only PE (processing element) 0 or a selected number of PEs communicate with other executable files like the master or the PVM daemon. To change the default communication behavior to decrease the number of socket connections made on the system, use the `PVM_PE_LIST` environment variable to specify which PEs should communicate through sockets. The `PVM_PE_LIST` environment variable specifies which processing elements can communicate with the PVM daemon. You can obtain the PE number for a task or process by calling the `pvm_get_PE` () or `my_pe` () functions. Set the environment variable as in the following examples:

For `csh`(1):

```
setenv PVM_PE_LIST 0, 4, 8, 12
```

or

```
setenv PVM_PE_LIST all
```

(default)

For `ksh`(1):

```
export PVM_PE_LIST=all
```

(default)

> **Note:** The default behavior of the `PVM_PE_LIST` environment variable in the UNICOS implementation is different from that on UNICOS/mk systems. In the UNICOS/mk implementation of PVM, by default, only PE 0 can communicate with the PVM daemon in heterogeneous programs.

You should also consider using Autotasking instead of message passing whenever your application is run on a UNICOS system. The communications overhead for Autotasking is orders of magnitude less than that for sockets, even on the same system, so you might be better off having only one fully autotasked PVM process on the UNICOS system. In many cases, you might be able to achieve this simply by invoking the appropriate compiler options and sending a larger file of input data to the PVM process on the UNICOS system.

2.3.9.2 Running PVM in Stand-alone Mode on UNICOS Systems

The PVM stand-alone mode of operation allows you to run UNICOS/mk applications on UNICOS systems. Not all UNICOS/mk applications are appropriate to run on UNICOS systems (because of size limitations, for example), but for those that are appropriate, several extensions have been added to facilitate porting.

Table 4 lists and describes the UNICOS extensions that are supported in stand-alone mode. When the term *PE* is used in this table, it refers to PVM processes or tasks that were spawned by using multitasking. For more information on the functions described in Table 4, see the appropriate man pages.

Table 4. UNICOS Extensions for Stand-alone Mode

Extension	Description
`pvm_get_PE ()`	A function that returns the PE number associated with a `pvm_tid` task identifier.
`barrier ()`	An optimized barrier function that can be used to create a barrier between multitasked PEs.
`_my_pe ()`	A function that returns the PE number of the PVM task and calls the task (similar to the intrinsic function available on UNICOS/mk systems). This function is documented in the `my_pe`(3) man page.
Global group	A predefined group that consists of all members of the multitasking group. This can be used with communication and synchronization between multitasked PEs. The variable `PVMALL` is declared in the `fpvm3.h` function, as it is for UNICOS/mk PVM applications. The concept of a predefined global group also exists on UNICOS/mk systems.
`_num_pes ()`	A function that returns the total number of PEs in the program (similar to the function available on UNICOS/mk systems). This function is documented in the `num_pes`(3) man page.
PE number	A PE number. Most existing UNICOS/mk PVM applications are written to use PE numbers to identify tasks for communication. To aid in porting UNICOS/mk applications to a UNICOS system, PE numbers can be used in place of `pvm_tid` task identifiers in many of the PVM functions. Functions that support PE numbers document this in their man pages.

2.3.9.3 Running PVM on UNICOS/mk Systems

The UNICOS/mk implementation of PVM can be used in stand-alone or distributed mode. In stand-alone mode, PVM is used only to communicate among processing elements (PEs) within the same partition. In this mode, the PVM daemon is not required; you can simply execute your program. For more information on using PVM in stand-alone mode, see Chapter 3, page 43. For more information on using PVM in distributed mode, see Section 3.5, page 58.

2.3.9.4 Using NQE

PVM applications can be run simultaneously. In previous releases, the same PVM daemon was used for all applications. If PVM applications are run as an NQS job, each application uses a unique PVM daemon. This can eliminate resource conflicts caused by both applications requesting PVM services. Because the daemon is part of the NQS job, resource limits associated with a job apply to processes spawned by the PVM daemon.

A site administrator can configure a batch job queue for PVM jobs. This enables the checking of resources on multiple nodes before a job is initiated.

2.3.9.5 Using Load Balancing

PVM supports load balancing for `pvm_spawn ()` calls on UNICOS systems. Support for this feature is deferred on UNICOS/mk and IRIX systems. PVM has been modified to request ratings of the eligible hosts. This feature is available only with NQE, but PVM does not have to be run as an NQS job to use this feature.

A `pvm_spawn ()` call proceeds through the logic of checking user-specified parameters, such as architecture. After a list of hosts has been identified, the load balancing server is asked to rate these hosts by evaluating a policy. Tasks are then placed, using a procedure based on a percentage of the total host ratings instead of a round-robin procedure. If the current host receives a large rating, all tasks can be started on that host. This allows hosts to be specified in the host file but not used if their current system load is large or a better host is available.

Specifying the `-h` option on the `pvmd3`(1) command when starting the PVM daemon causes the load balancing server to be used to create the virtual machine. If a host file is specified, the options are read. The NQE load balancing server is queried for a list of available hosts. The hosts specified in the host file but not marked as available are not used. Hosts not specified in the host file are added with default options. If no host file is specified, all of the available hosts are used with default options.

Automatic creation of the virtual machine allows the user to submit a PVM job without configuring the machine.

2.3.9.6 Using PVM Direct Routing

If two PVM processes are going to be doing any significant amount of communication, you should probably use PVM direct routing. With a normal transfer, a message goes from one PVM user process, to the PVM daemon on the local system, to the PVM daemon on the remote system, and finally to the PVM user process on the remote system. With direct routing, a message goes directly from one PVM task to the other. A significant performance gain is possible, and the gain increases for larger messages. For more information on setting the associated PVM option, see the `pvm_setopt`(3) man page.

2.3.9.7 Using Large Messages

Socket communication in PVM uses a default maximum packet size of 32 Kbytes. When sending large messages, you can increase bandwidth significantly by using the `pvm_setopt ()` call to set `PvmFragSize`. Although underlying services might have lower limits, PVM's upper limit for `PvmFragSize` is 1 Mbyte. On UNICOS systems, you can change the TCP window shift size by using the `pvm_setopt ()` call to set `PvmWinShift`. Valid values for `PvmWinShift` are between 1 and 16. Values above 4 show marginal performance improvement. This can increase the amount of data in transmission.

2.3.9.8 Avoiding XDR Conversion

By default, PVM automatically performs eXternal Data Representation (XDR) conversion when transferring data. This very powerful feature adds to the utility of PVM. Unfortunately, this feature has an adverse effect on performance, because the conversion method is very slow and inefficient.

In many applications, you can work around this limitation by using one of the following techniques:

- Using `PvmDataRaw`
- Transferring bytes of data
- Performing data conversion in the application (*user-controlled conversion*)
- Running PVM in stand-alone mode (see Section 2.3.9.2, page 30)

2.3.9.8.1 Specifying the `PvmDataRaw` Value

The `pvm_initsend`(3) function takes an argument that specifies how data should be encoded. Specifying the `PvmDataRaw` value indicates that no data conversion should take place. When the `pvm_send`(3) call is made, PVM verifies that the two systems share a common data format and aborts the transfer of the message if they do not.

When appropriate, this technique is the best one to use. However, it is useful only when the two processes are running on systems with the same data format.

2.3.9.8.2 Byte Transfer

When data is transferred as bytes, no data conversion occurs. The `pvm_pkbyte`(3) and `pvm_upkbyte`(3) functions bypass data conversion because byte data is defined to be untyped. However, you must pass byte counts rather than element counts. As a result, this technique is more prone to programming error than the other two described in this section, especially for Fortran codes.

This technique is useful when the two processes are running on systems with the same data formats or the data format is one that does not require sophisticated conversions (such as packed integer data).

2.3.9.8.3 User-controlled Conversion

The most complex technique is user-controlled conversion; that is, performing data conversion in the application. UNICOS libraries offer a number of very high-performance data conversion functions that convert between Silicon Graphics formats and IEEE, IBM, DEC, and other formats.

For example, one process (perhaps a workstation) could use `pvm_pkbyte`(3) to pack the data; the receiving process (the Silicon Graphics system) could use `pvm_upkbyte`(3) to unpack the data and then call the appropriate function to convert it.

User-controlled conversion requires programmer care to ensure that the two processes know the format of the data blocks, and it also requires byte counts rather than element counts.

This technique is useful when the transfer is occurring between a Silicon Graphics system and another system for which data conversion functions are available. (For descriptions of conversion functions, see the *Application Programmer’s Library Reference Manual*.)

2.3.10 Shared Memory PVM Limitations

Note: Information in this section is for UNICOS systems only.

Emulating a shared memory environment with the use of Cray multitasking software might provide unexpected program behavior. The goal is to preserve the original behavior as much possible. However, it is not efficient or productive to preserve completely the original PVM behavior in a multitasked environment. The intent is to document possible changes in behavior. For example, changes in behavior might occur with the use of signals; therefore, it is not recommended that signals be used with the shared memory version of PVM.

`PvmDataDefault` and `PvmDataRaw` packing are equivalent in the shared memory implementation of PVM. In stand-alone mode, data conversion is not necessary because the executable file never communicates outside of the UNICOS machine. Because the shared memory implementation of PVM running in standard mode does not handle data conversion, multitasked executable files currently can communicate only between UNICOS machines of the same architecture type. Communication to other architectures can still be achieved, however, through the master or nonmultitasked executable files that comprise the program.

The shared memory implementation of PVM supports the running of only 32 PVM tasks within a multitasking group. Running with more than the number of physical CPUs available on the UNICOS system will begin to degrade performance because PVM tasks must share CPU resources.

2.4 Data Types on UNICOS Systems

This discussion of how PVM data types are implemented assumes that you are familiar with the functions used to pack and unpack data. For more information about these functions, see Section 4.9, page 68, and Section 4.10, page 69.

Data type support is different for each system. Systems that support both 32-bit and 64-bit data types map easily into the PVM data types.

Table 5, page 35, presents basic information about data types on UNICOS systems.

Table 5. Data Types On UNICOS Systems

Data characteristics	C functions	Fortran names
8 bits, not typed	`pvm_pkbyte`	`BYTE1`
64 bits, signed integer	`pvm_pklong`, `pvm_pkint`, `pvm_pkshort`	`INTEGER4`
64 bits, unsigned integer	`pvm_pkulong`, `pvm_pkuint`, `pvm_pkushort`	Not applicable
64 bits, floating-point	`pvm_pkdouble`, `pvm_pkfloat`	`REAL8`

Data characteristics	C functions	Fortran names
Two 64 bits, floating-point	`pvm_pkdcplx`, `pvm_pkcplx`	`COMPLEX16`
Null-terminated character string	`pvm_pkstr`	Not applicable
Fortran character constant or variable	Not applicable	`STRING`

2.4.1 Fortran Data Types

Table 5 does not contain some Fortran type names, such as `INTEGER2`, `REAL4`, and `COMPLEX8`. These map to `INTEGER4`, `REAL8`, and `COMPLEX16`, respectively. To avoid confusion and to ease porting to other systems, you should use these only with the greatest of care.

2.4.2 64-bit Integer Usage

The Cray MPT implementation of PVM for UNICOS systems does not support the `INTEGER8` specification. On UNICOS systems, you must specify `INTEGER4` for UNICOS integers.

When `PvmDataDefault` packing is used, XDR converts data into a common format. XDR retains only 32 bits of precision for integer data; therefore, packing 64-bit integers results in a loss of the upper 32 bits of precision.

If you want all 64 bits of accuracy, use `PvmDataRaw` packing or specify untyped byte packing for `PvmDataDefault`.

2.5 Data Types on UNICOS/mk Systems

This section describes how PVM data types are implemented on UNICOS/mk systems. This discussion assumes that you are familiar with the functions used to pack and unpack data; for more information, see Section 4.9, page 68, and Section 4.10, page 69.

Table 6 presents basic information about data types available on UNICOS/mk systems.

Table 6. Data Types on UNICOS/mk Systems

Data characteristics	C functions	Fortran names
8 bits, not typed	`pvm_*pkbyte`	`BYTE1`
64 bits, signed integer	`pvm_*pklong`, `pvm_*pkint`	`INTEGER8`
64 bits, unsigned integer	`pvm_*pkulong`, `pvm_*pkuint`	Not applicable
32 bits, signed integer	`pvm_*pkshort`	`INTEGER4`
32 bits, unsigned integer	`pvm_*pkushort`	Not applicable
64 bits, floating-point	`pvm_*pkdouble`	`REAL8`
32 bits, floating-point	`pvm_*pkfloat`	`REAL4`
(Two) 64 bits, floating-point	`pvm_*pkdcplx`	`COMPLEX16`
(Two) 32 bits, floating-point	`pvm_*pkcplx`	`COMPLEX8`
Null-terminated character string	`pvm_*pkstr`	Not applicable
Fortran character constant or variable	Not applicable	`STRING`

2.5.1 16-bit Fortran Data Types

The Fortran name `INTEGER2` is implemented and maps into the same data characteristics as `INTEGER4`. Use of 16-bit data types is not recommended because the UNICOS/mk system does not support these data types.

2.5.2 32-bit Fortran Data Types

On the UNICOS/mk system, PVM supports 32-bit Fortran data types. This support is implemented in PVM regardless of whether your Fortran compiler supports 32-bit data types. If you are not using such a compiler and specify one of these data types (`INTEGER4`, `REAL4`, or `COMPLEX8`), you will get incorrect results.

2.5.3 64-bit Integer Data

In the UNICOS and public domain versions of PVM, data conversion of integers is limited to 32 bits of accuracy. The UNICOS/mk version handles 64-bit integers in a manner that is compatible and interoperable with the network version. If you pack 64-bit integers into a `PvmDataDefault` block, only the low-order 32 bits of each value are packed. PVM checks the high-order 32 bits of each value; if they contain significant data, the pack call sends a `PvmLostPrecision` error. (This checking can be turned off; see `PVM_CHECKING` in Table 11, page 46.)

If you want 64 bits of accuracy, you can use `PvmDataRaw` packing or specify untyped byte packing for `PvmDataDefault`.

2.6 Data Types on IRIX Systems

This section describes how PVM data types are implemented on IRIX systems. This discussion assumes that you are familiar with the functions used to pack and unpack data; for more information, see Section 4.9, page 68, and Section 4.10, page 69.

Table 7 and Table 8 present basic information about data types on IRIX systems.

Table 7. N32 ABI Library Data Types on IRIX Systems

Data characteristics	C functions	Fortran names
8 bits, not typed	`pvm_pkbyte`	`BYTE1`
16 bits, signed integer	`pvm_pkshort`	`INTEGER2`
32 bits, signed integer	`pvm_pkint`, `pvm_pklong`	`INTEGER4`
16 bits, unsigned integer	`pvm_pkushort`	Not applicable
32 bits, unsigned integer	`pvm_pkuint`, `pvm_pkulong`	Not applicable
32 bits, floating-point	`pvm_pkfloat`,	`REAL4`
64 bits, floating-point	`pvm_pkdouble`	`REAL8`
Two 32 bits, floating-point	`pvm_pkcplx`	`COMPLEX8`
Two 64 bits, floating-point	`pvm_pkdcplx`	`COMPLEX16`

Data characteristics	C functions	Fortran names
Null-terminated character string	`pvm_pkstr`	Not applicable
Fortran character constant or variable	Not applicable	`STRING`

Table 8. 64 ABI Library Data Types on IRIX Systems

Data characteristics	C functions	Fortran names
8 bits, not typed	`pvm_pkbyte`	`BYTE1`
16 bits, signed integer	`pvm_pkshort`	`INTEGER2`
32 bits, signed integer	`pvm_pkint`	`INTEGER4`
64 bits, signed integer	`pvm_pklong`	Not applicable
16 bits, unsigned integer	`pvm_pkushort`	Not applicable
32 bits, unsigned integer	`pvm_pkuint`	Not applicable
64 bits, unsigned integer	`pvm_pkulong`	Not applicable
32 bits, floating-point	`pvm_pkfloat,`	`REAL4`
64 bits, floating-point	`pvm_pkdouble`	`REAL8`
Two 32 bits, floating-point	`pvm_pkcplx`	`COMPLEX8`
Two 64 bits, floating-point	`pvm_pkdcplx`	`COMPLEX16`
Null-terminated character string	`pvm_pkstr`	Not applicable
Fortran character constant or variable	Not applicable	`STRING`

2.7 Environment Variables

To customize your PVM environment, you can use the environment variables described in this section. The variables are grouped into variables supported on IRIX systems only and variables supported on UNICOS, UNICOS/mk, and IRIX systems. Chapter 3, page 43, describes the environment variables that are supported by UNICOS/mk systems only.

2.7.1 Setting Environment Variables on IRIX Systems

This section provides a table of environment variables you can set for IRIX systems only.

Table 9. Environment Variables on IRIX Systems

Variable	Description	Default
`PVM_SHMEM_DIR`	Directory location of the POSIX shared memory files.	`/usr/tmp` (Only valid for SGIMP64 and SGIMP64mips3 architecture types)
`PVMBUFSIZE`	Specifies the size of the shared memory buffer for each task and daemon.	1 Mbyte
`PVM_VMID`	Sets the virtual machine identification (VMID) number for the host. This environment variable allows a host to be included in more than one virtual machine by using one `pvmd3` command per virtual machine per host. The virtual machine number is appended to the file name of the PVM log and daemon socket files, so that they appear as `pvml.`*uid.vmid* and `pvmd.`*uid.vmid*. The previous name of this variable is `PVMJID`. This name is supported in the MPT 1.3 release, but will not be supported in subsequent releases. **Note:** This environment variable prevents IRIX PVM from interoperating with any implementation other than Silicon Graphics IRIX PVM implementations.	0

2.7.2 Setting Environment Variables on UNICOS, UNICOS/mk, and IRIX Systems

This section provides a table of environment variables you can set for UNICOS, UNICOS/mk, and IRIX systems.

Table 10. Environment Variables on UNICOS, UNICOS/mk, and IRIX Systems

Variable	Description	Default
`PVM_ROOT`	Specifies the path where PVM libraries and system programs are installed. For PVM to function, this variable must be set on each PVM system.	Set automatically when you load the `mpt` module to access the Message Passing Toolkit software
`PVM_EXPORT`	Names the environment variables that a parent task exports to its children by using the `pvm_spawn`(3) function. Multiple names must be separated by a colon.	None
`PVM_DEBUGGER`	Specifies the debugger script to use when `pvm_spawn`(3) is called with `PvmTaskDebug` set.	`$PVM_ROOT/lib/debugger`
`PVM_DPATH`	Specifies the path of the `pvmd3`(1) command or the startup script. If you use a shell (such as `.kshrc`) that does not automatically execute a startup script that sets `PVM_ROOT` on added hosts, you can set `PVM_DPATH` to the full or relative path of the `pvmd` startup script, such as `$PVM_ROOT/lib/pvmd`. This startup script automatically sets `PVM_ROOT`.	`$PVM_ROOT/lib/pvmd`. You can override this setting by using the `dx=` *loc* option in the host file.
`PVM_POLICY`	Specifies the NQE policy used for load balancing. For more information on specifying policies, see *NQE Administration*. **Note:** Support for this environment variable is deferred on UNICOS/mk and IRIX systems.	`PVM`

Variable	Description	Default
`NLB_SERVER`	Specifies the location of the NQE load balancer. This host is known as the *master server*. Your system administrator might have this set automatically in the `nqeinfo` file. If NQE load balancing is enabled on your system, it is used automatically by PVM. To disable NQE load balancing for PVM applications, set the `NLB_SERVER` environment variable to 0. For more information, see the *NQE User's Guide*. **Note:** Support for this environment variable is deferred on UNICOS/mk and IRIX systems.	Value in the `nqeinfo` file
`PVM_RSH`	Specifies that an alternative remote shell command, such as `krsh` (a Kerberos version of `rsh`), can be selected. `PVM_RSH` can specify the full path or relative path to the alternative remote command.	IRIX: If using Array Services, `/usr/sbin/arshell`. If not using Array Services, `/usr/bsd/rsh`. UNICOS or UNICOS/mk: `/usr/ucb/remsh`.
`PVM_SLAVE_STARTUP_TIMEOUT`	Specifies the length of time that the master daemon will wait for a slave daemon to make contact after the slave daemon is started.	60 seconds

UNICOS/mk Implementation [3]

This chapter describes aspects of PVM that are specific to UNICOS/mk systems. On a UNICOS/mk system, which contains up to 2048 processing elements (PEs), some subset of this number of PEs is assigned to a job running on the system. Those PEs are collectively known as a *partition*.

The UNICOS/mk implementation of PVM can be used in either or both of the following modes:

- *Stand-alone mode*, in which PVM is used for communication (PE-to-PE) within the partition.
- *Distributed mode*, in which PVM is used to communicate outside the partition.

There is one PVM library that is part of the MPT product environments for the UNICOS/mk system. When the UNICOS/mk executable file is initiated, it determines whether it is being used in distributed mode and performs the proper setup. If it determines that it is not being used in distributed mode, certain PVM functions are not available and return errors if called.

3.1 Features and Differences

This section summarizes special features that can be found in the UNICOS/mk version of PVM and notes differences between it and the other versions. These features are also documented in the applicable PVM man pages.

3.1.1 PE Number

Most existing UNICOS/mk applications and algorithms are written to use PE numbers for communication. Standard PVM notation used only the concept of a PVM task identifier (`pvm_tid`, whose internal representation is subject to change). To simplify programming, the UNICOS/mk version lets you use PE numbers in place of `pvm_tid`s in many of the PVM functions. An extra function, `pvm_get_PE`(3), returns the PE number associated with a `pvm_tid`.

3.1.2 Global Group

PVM supports the concept of *dynamic groups*, in which tasks can join and leave groups at any time. The barrier and broadcast functions use these groups for collective synchronization and communications. On a UNICOS/mk system, a static, well-defined group consisting of all the tasks (or PEs) in the partition is referred to as the *global group*. To simplify programming, PVM has essentially predefined this group by permitting a null name (or, in C, a null `char` pointer) to be used to refer to this global group. (The Fortran PVM include file, `fpvm3.h`, contains a declaration of a null character variable, `PVMALL`.) PVM uses some key optimizations to carry out barriers and broadcasts for the global group.

3.1.3 Obtaining PE Numbers

UNICOS/mk applications can use PVM calls to obtain their own PE number. From C these calls are as follows:

```
my_pe = pvm_get_PE (pvm_mytid());
```

From Fortran the calls are as follows:

```
CALL PVMFMYTID (MYTID)
CALL PVMFGETPE (MYTID,MYPE)
```

3.1.4 Number of PEs

UNICOS/mk applications can use PVM calls to obtain the number of PEs in the partition. From C this is as follows:

```
n_pes = pvm_gsize(0);
```

From Fortran the call is as follows:

```
CALL PVMFGSIZE (PVMALL, NPES)
```

The variable `PVMALL` is declared in `fpvm3.h`.

3.1.5 `PvmDataInPlace` Semantics

The UNICOS/mk version of PVM treats data buffers packed using `PvmDataInPlace` encoding differently than the network version does. In the UNICOS/mk version, such data must not be reused until the data has been unpacked by the receiving PE. You are responsible for any additional synchronization or communication required to ensure this coordination.

3.2 Using Environment Variables to Change Default Settings

You can control a number of features and settings in PVM. The default behavior and settings of PVM may not be suitable for all or part of some applications, and you may wish to change them. In general, you can set options in two ways:

- Many options can be set by using the `pvm_setopt`(3) function. This function allows an option to be set for a specific PE or to be changed dynamically during execution of an application. For example, if the `pvm_parent`(3) function is called to see if the application is being used in distributed mode, the following code sequence ensures that a return code of `PvmNoParent`, which is considered an error, does not cause the program to abort or print out an error message:

  ```
  oldvalue = pvm_setopt (PvmAutoErr, 0);
  parent_id = pvm_parent ();
  (void) pvm_setopt (PvmAutoErr, oldvalue);
  ```

- Many options can be set by using the UNICOS/mk environment variables without changing source code. These take effect with PVM initialization and apply to the application as a whole.

While many options can be set by using either mechanism, some can only be set using one mechanism or the other. This section describes those that you can set by using UNICOS/mk environment variables. Table 11, page 46, lists the UNICOS/mk environment variables. For more information about the `pvm_setopt`(3) function, use the `man`(1) command to view the man page online.

When setting an environment variable, you must ensure that it is available for the UNICOS/mk executable file. If you are using the UNICOS/mk version in stand-alone mode, this means that the environment variable must be set before the executable file is run:

```
% setenv PVM_TRACE 7
% ./t3e.out
```

If you are using PVM in distributed mode, the PVM daemon starts the UNICOS/mk executable file. Therefore, you must set the environment variable before the daemon is started, as follows:

```
% setenv PVM_TRACE 7
% pvmd3 hostfile
```

Remember, it is the UNICOS/mk daemon, not the task that calls pvm_spawn(3), that starts the UNICOS/mk executable file.

The PVM_ROOT environment variable specifies the path at which PVM libraries and system programs are installed. For PVM to function, this variable must be set on each PVM system. On UNICOS/mk systems, $PVM_ROOT is set for you automatically when you load the mpt module to access the MPT software.

Table 11. UNICOS/mk Environment Variables

Variable	Description	Default
PVM_AUTO_ERR	Sets the PVM error-handling value, which is equivalent to the PvmAutoErr option in pvm_setopt(3). Setting this value with PVM_AUTO_ERR lets you do so without changing your source.	1 (error reporting on)
PVM_CHECKING	Certain common PVM operations run the risk of losing data. By default, PVM performs a check to avoid this problem. While the cost of this check is not prohibitive, it can have an impact on performance, and might be unnecessary for your application. The PVM_CHECKING environment variable lets you control whether the check is performed. This control is at a very gross level: either the check is performed throughout the entire program or it is not performed at all. When PvmDataDefault encoding is used for packing 64-bit integer data, only the low-order 32 bits are packed. By default, PVM checks whether any of the truncated high-order bits contained significant data and generates an error (PvmLostPrecision) if they did. If you set PVM_CHECKING to 0, this check is not performed. If you set PVM_CHECKING to 1 (the default setting), the check is performed.	1 (Check is performed)
PVM_DATA_BUFFERS	Sets the initial and incremental number of send buffers. For more information on send buffers, see Section 3.3, page 48.	Initial: 0 blocks; incremental: 1 block

Variable	Description	Default
`PVM_DATA_MAX`	Sets the integer number of the maximum number of bytes in an initial message. The specified value must be a multiple of 8. When a message is sent with PVM, the library sends a header and a relatively small amount of data in an initial message. The default size for this data is 4096 bytes. Messages that contain more than this amount of data must transfer the data later in a second, slower transfer. By increasing the amount of data that can be transferred with the initial message, you can reduce communications overhead. The value of `PVM_DATA_MAX` represents memory that is taken up by internal message pools and allocated for each message structure active in the system (whether or not the memory is actually used for a given message). The larger the value, the more memory that is used by PVM and unavailable to the application. The smaller the value, the more messages that will require a second transfer. `PVM_DATA_MAX` has a particularly significant impact on the performance of messages broadcast to multiple tasks, due to the way these are implemented on the UNICOS/mk system. If a broadcast is used in a time-critical portion of code, you may want to verify that `PVM_DATA_MAX` is at least as large as the message being broadcast.	4096 (The default value is in the description)
`PVM_MAXGTIDS`	Changes the maximum number of tasks that can join a group. For information about the out-of-resources error, `PvmOutOfResGmems`, see Section 3.4.3, page 58.	`sysconf(_SC_CRAY_NPES)` (Number of PEs in application)
`PVM_MAX_PACK`	Sets the initial and incremental data block sizes. For information about setting this variable, see Section 3.3.3.3, page 52, and Section 3.3.3.4, page 52.	Initial: 4096 bytes; incremental: 4096 bytes
`PVM_PE_LIST`	Lists the virtual PE numbers within a partition that can communicate with the daemon. Either a comma-separated list of virtual PE numbers or `all` can be specified. If `all` is used, all PEs in a partition can communicate with the daemon.	Only PE 0 communicates with the daemon.

Variable	Description	Default
`PVM_RETRY_COUNT`	Sets the number of times that PVM retries sending a message to another PE before giving up and returning a `PvmOutOfResSMP` error. For more information, see Section 3.4.1, page 56.	500
`PVM_SM_POOL`	When PVM is started up, it allocates a pool of shared memory for use in message passing. This pool represents space used to buffer message headers and small messages while the receiving PE is doing computations or I/O. Each entry or message uses `PVM_DATA_MAX` plus 32 bytes of memory. The `PVM_SM_POOL` environment variable sets the integer number for the number of messages in the pool for each PE. For information about the out-of-resources error, `PvmOutOfResSMP`, see Section 3.4.1, page 56.	The larger of the following values: • Two times the number of PEs • 10
`PVM_TOTAL_PACK`	Establishes the upper limit on memory allocated for send buffer data blocks. For information about setting this variable, see Section 3.3.3.5, page 53.	999,999,999
`PVM_TRACE`	Sets a mask of trace options, equivalent to the `PvmTraceOpts` options in `pvm_setopt`(3). Using `PVM_TRACE` to set these options lets you do so without changing the source. This environment variable controls only the collection of trace data, not its output. The `pvm_disptrace`(3) function is used to display trace data.	All tracing is off.

3.3 Buffer Memory Management

When `PvmDataDefault` and `PvmDataRaw` encoding is used, PVM allocates and uses blocks of memory on the sending PE. (These blocks are referred to as *send buffers*.) By default, this allocation and usage is transparent to your application; that is, you should not have to do anything special. However, if your application is trying to optimize its use of memory, you may need to understand how PVM uses memory, and you may want to control PVM memory usage. This section discusses these topics.

3.3.1 Basic Design

The design of buffer memory is based on the following:

- By default, all send buffer space is dynamically allocated in the following manner:
  - Memory is allocated only if needed.
  - Only the amount of memory needed is allocated.
  - Portions of memory are freed once they are no longer needed.
- By using environment variables, you can control initial allocation of send buffers.
- By using environment variables or `pvm_setopt`(3) calls, you can change the amount of additional memory allocated for each send buffer, and control or prohibit incremental memory units when even more memory is required.
- You can specify a total limit on the amount of memory allocated at any one time for the send buffers.

Send buffers are never freed by PVM. Once allocated and used, they are kept for later use. However, any incremental memory allocated for a send buffer is freed as soon as it is no longer needed.

3.3.2 Simple Scenario, Part 1

The scenario in Table 12 shows PVM memory use, using the default settings.

Table 12. Default Settings for Buffer Memory Management

User call	PVM action	Memory use in bytes (sending PE)
`pvm_initsend (PvmDataRaw);`	Allocates send buffer.	4096
`pvm_pkbyte (...32...);`	Copies data. (4064 bytes are free.)	4096
`pvm_pkbyte (...32000...);`	Copies 4064 bytes. Allocates 32000 – 4064 = 27936 bytes. Copies remaining data.	32,032

User call	PVM action	Memory use in bytes (sending PE)
`pvm_pkbyte (...32...);`	Allocates 4096 bytes. Copies data. (4064 bytes are free.)	36,128
`pvm_pkbyte (...40...);`	Copies data. (4024 bytes are free.)	36,128
`pvm_send (...);`	Sends message.	36,128
`pvm_recv (...);`	Receives message.	36,128
Final `pvm_upkbyte` by receiving PE for message or `pvm_recv` call for next message	Frees incremental data blocks. Returns buffer to free list.	4096

This scenario shows how PVM allocates memory for send buffers. Although 36,128 bytes were allocated, only 32,104 were actually used. The 4096 bytes allocated in the second incremental allocation were used for only 72 bytes.

3.3.3 Controlling Memory Use

The following parameters are available for controlling send buffer memory use:

- Initial number of send buffers
- Send buffer increment
- Send buffer initial size
- Send buffer increment size
- Total memory use

You can set all five parameters by using environment variables, which take effect at PVM initialization time. Four of the five can also be set by calling `pvm_setopt`(3), which changes the settings dynamically at run-time. (The fifth parameter, initial number of send buffers, affects an initialization time function, and so a run time change would have no effect.) You can call the `pvm_getopt`(3) function to obtain the current settings for all five parameters.

Only three environment variables are needed to set the five parameters because two of these variables let you set either one or two parameters at once.

By using an environment variable, you set the value for all PEs at once. By calling `pvm_setopt`(3), you can set different values for different PEs, or you can change a value during the execution of the program. You can, of course, combine the two mechanisms by using the environment variables to set the default values and `pvm_setopt`(3) to change specific cases.

The following sections discuss how you can use and set these parameters.

3.3.3.1 Initial Number of Send Buffers

During initialization time, PVM allocates an initial number of send buffers. The default is 0; that is, no send buffers are allocated initially. In this case, as soon as you call the `pvm_initsend`(3) function with the `PvmDataDefault` or `PvmDataRaw` option, a new send buffer is dynamically allocated. This requires library calls and possibly an operating system call, and thus is expensive in time. Alternatively, you can initially allocate some send buffers, perhaps enough to avoid having to dynamically allocate any additional buffers.

To set the initial number of send buffers, enter the `PVM_DATA_BUFFERS` environment variable as follows:

`setenv PVM_DATA_BUFFERS <`*number*`>`

In the following example, the `PVM_DATA_BUFFERS` setting tells PVM to initially allocate 10 send buffers:

```
setenv PVM_DATA_BUFFERS 10
```

The `pvm_setopt`(3) function does not support this parameter. You can call `pvm_getopt`(3) with the `PvmDataBuffers` option to find out the value of `PVM_DATA_BUFFERS`.

3.3.3.2 Send Buffer Increment

Whenever PVM dynamically allocates a new send buffer, it makes library calls to allocate memory for a specified number of send buffers. The default is 1; that is, PVM allocates enough memory for a single new send buffer. This process is expensive in time because PVM must make another set of library calls to allocate more memory each time a new buffer is needed.

You can amortize the cost of the library calls by using the send buffer increment parameter. This parameter setting tells PVM to allocate enough memory for a specified number of additional buffers each time it needs to allocate memory for a single one.

This parameter can also tell PVM not to allocate additional memory for send buffers. By setting the initial number of send buffers to some number and setting the increment to 0, you can fix the number of send buffers allocated by

PVM. In this case, if PVM runs out of send buffers, your application receives a `PvmOutOfResBuf` error.

The send buffer increment is the second option on the `PVM_DATA_BUFFERS` environment variable. To set this parameter, enter `PVM_DATA_BUFFERS` as follows:

`setenv PVM_DATA_BUFFERS` *number+increment*

In the following example, the `PVM_DATA_BUFFERS` setting tells PVM to initially allocate 10 send buffers and to allocate 4 more at a time if more buffers are needed, up to a total of `PVM_TOTAL_PACK`:

```
setenv PVM_DATA_BUFFERS 10+4
```

You can use the `PvmDataBuffersIncr` option with `pvm_setopt`(3) to change the setting dynamically. You can also use this option with `pvm_getopt`(3) to see the send buffer's increment setting.

3.3.3.3 Send Buffer Initial Size

Each send buffer contains an initial block of memory for use in packing data. The default is 4096 bytes. If more is needed, PVM makes library calls to allocate an additional block. If less is needed, the difference is wasted. If you know that most messages in your code are of a specific size, you can set this parameter to that size to avoid wasting memory or allocating additional blocks.

To set the send buffer initial size, enter the `PVM_MAX_PACK` environment variable as follows:

`setenv PVM_MAX_PACK` *initial*

In the following example, the `PVM_MAX_PACK` setting tells PVM to initially allocate 16,384 bytes of memory for each send buffer:

```
setenv PVM_MAX_PACK 16384
```

You can use the `PvmMaxPack` option with `pvm_setopt`(3) to change the setting dynamically. You can also use this option with `pvm_getopt`(3) to see the send buffer initial size.

3.3.3.4 Send Buffer Increment Size

When PVM dynamically allocates an additional block of memory, it uses a minimum allocation size. The default is 4096 bytes. If PVM needs less than this amount of memory, it allocates the minimum size. If PVM needs more than this minimum size, it allocates what it needs.

The send buffer increment size parameter enables you to avoid multiple allocations of blocks that are only a few words in length. For example, if most of your messages fit within 4096 bytes, but you have one large message that requires a total of 164,096 bytes, you could set this parameter to 160,000 bytes.

This parameter can also be set to 0 to tell PVM that it must not allocate additional memory blocks. In this case, if the data fails to fit into the initial block, PVM returns a `PvmTooMuchData` error to your application.

The send buffer increment size is the second option on the `PVM_MAX_PACK` environment variable. To set this parameter, enter `PVM_MAX_PACK` as follows:

`setenv PVM_MAX_PACK` *initial+increment*

In the following example, the `PVM_MAX_PACK` setting tells PVM to initially allocate 4096 bytes of memory for each send buffer, but, if more is needed, to allocate a block no smaller than 160,000 bytes:

```
setenv PVM_MAX_PACK 4096+160000
```

You can use the `PvmMaxPack` option with `pvm_setopt`(3) to change the setting dynamically. You can also use this option with `pvm_getopt`(3) to see the send buffer increment size.

3.3.3.5 Total Memory Use

PVM tracks the amount of memory allocated for data blocks, both initial and incremental blocks. There is no set default; you are limited only by the available memory in the PE.

The total memory use parameter establishes a limit for the amount of memory allocated. If PVM exceeds this limit, it returns a `PvmMemLimit` error to your application.

This parameter does not reflect total memory usage by PVM, but only the data block allocation associated with send buffers. For many applications, this is the predominant source for PVM memory usage.

To set the total memory use parameter, enter the `PVM_TOTAL_PACK` environment variable as follows:

`setenv PVM_TOTAL_PACK` *limit*

In the following example, the PVM_TOTAL_PACK setting tells PVM to use no more than 1,048,576 bytes of memory at any time for send buffer data blocks:

```
setenv PVM_TOTAL_PACK 1048576
```

You can use the PvmTotalPack option with pvm_setopt(3) to change the setting dynamically. You can also use this option with pvm_getopt(3) to see total memory use. To see how much memory is remaining from the current limit, use the PvmTotalPackLeft option with pvm_getopt(3).

3.3.4 Simple Scenario, Part 2

In the original scenario (Section 3.3.2, page 49), 36,128 bytes of buffer memory were allocated, but only 32,104 were actually used. Memory use could be made most efficient by using PvmDataInPlace encoding, which avoids PVM buffer allocation altogether. But this change may require some additional synchronization within the program, and thus it may not be desirable.

Next in order of simplicity, you could move the large pvm_pkbyte(3) call (with 32,000 bytes) to the end. Consequently, the three small packs would go into the initial 4096 bytes, and just enough bytes would be allocated for the large pvm_pkbyte(3) call.

Instead (or in addition), the following PVM_MAX_PACK settings could be considered to more efficiently manage memory:

```
setenv PVM_MAX_PACK 32104
```

This setting ensures that all the memory needed is allocated with the send buffer. If all message traffic looked like this, this would be most efficient. By setting PVM_MAX_PACK to 32104+0, you could verify that no message exceeded this limit.

```
setenv PVM_MAX_PACK 4096+28008
```

This setting ensures that the first incremental memory allocation is sufficient for the remaining packs. If most messages fit into 4096 bytes, and the rest fit into 32,104 bytes, this setting limits normal memory use while avoiding unnecessary malloc(3) or free(3) calls for the large messages.

This scenario shows only the memory allocation for a single message. A real application has many messages of different sizes; therefore, while PVM_MAX_PACK settings might help this one message, they might have adverse effects on others.

If only one PE is sending a large message, another approach is to change the source code so that this PE calls `pvm_setopt`(3) once with `PvmMaxPack` and perhaps again with `PvmMaxPackIncr`, each set to the values indicated in the previous `setenv` commands, prior to packing and sending the large message. For example, you could call `pvm_setopt` with `PVM_MAX_PACK` set equal to 32,104, or you could call `pvm_setopt` with `PVM_MAX_PACK` set equal to 40 and call `pvm_setopt` with `PvmMaxPackIncr` set equal to 28,008.

3.4 Out-of-resource Errors

When running a PVM application on UNICOS/mk systems, you may receive out-of-resource errors. Receiving one of these errors, shown in Table 13, means that you have encountered a fixed limit within the PVM implementation.

Table 13. Out-of-resource Errors

Error	Fixed limit
`PvmOutOfResSMP`	A shared memory pool of messages used in sends
`PvmOutOfResBuf`	A preallocated set of data buffers used by `pvm_initsend`(3), `pvm_recv`(3), and related functions
`PvmOutOfResGmems`	A maximum number of tasks that can join a group

These limits are fixed for various reasons, but you can raise each of them. However, you should be careful about doing so for two reasons:

- Raising a limit causes PVM to allocate more memory, and this memory is not available for your application to use.
- Your application may not be using PVM efficiently. Making a simple code change may eliminate the error and also give you better performance.

Two of the out-of-resource conditions (`PvmOutofResSMP` and `PvmOutofResBuf`) might occur only occasionally, due to unusual timing circumstances. Instead of wasting memory to handle these unlikely situations, consider writing your application to accept these errors if they occur and to retry the action that caused the error until the action succeeds. For example, the following code fragment retries a send until it succeeds:

```
10 CONTINUE
   CALL PVMFSEND (OTHERPE, TAG, INFO)
   IF (INFO.EQ. PVMOUTOFRESSMP) GOTO 10
   IF (INFO.LT.0) CALL ABORT()
```

Out-of-resource errors often appear when you are increasing the number of processors being used or the size of the problem being solved. Several options are available for dealing with the limits you encounter. The following sections briefly discuss each limit, describe how to raise it, and identify ways to use PVM more efficiently.

3.4.1 PvmOutOfResSMP

A pool of memory is allocated in each PE to receive messages from other PEs. When a message is sent, the sending PE uses part of the pool on the receiving PE for the message. At the beginning of various PVM functions, a receiving PE checks for any messages in this pool and clears them out. If too many PEs try to send messages before a PE can clear out the pool, the pool becomes exhausted, and subsequent sends may fail with the `PvmOutOfResSMP` error.

By default, sends that detect this condition enter a retry loop, in which they delay briefly and then recheck the pool. This loop is performed `PVM_RETRY_COUNT` times (default is 500), and the `PvmOutOfResSMP` error is issued at the end of this count. You can adjust this limit up or down as described in Table 11, page 46. Many applications will find that increasing this count is sufficient to get by the error.

You can also adjust the number of entries in the pool. The default limit is twice the number of PEs or 10, whichever is larger. You can raise or lower this limit by using the `PVM_SM_POOL` environment variable, described in Table 11, page 46.

If you are hitting the pool entry limit, you may want to see if the receiving PE can be changed to call `pvm_recv`(3) or `pvm_nrecv`(3) sooner. This problem can occur if all PEs are broadcasting to each other and then trying to receive the results. By interspersing the broadcasts with the receives, you may avoid having to raise the limit.

You may also hit the pool entry limit if many messages are being sent to a PE that is busy doing some computation, waiting for I/O, or doing something else that keeps it from entering PVM. Increasing the limit allows such operations to proceed asynchronously; changing the code to operate more synchronously is another option.

3.4.2 `PvmOutOfResBuf`

The `PvmOutOfResBuf` error occurs only if you have set the send buffer increment parameter to 0 (see Section 3.3.3.2, page 51, for information on setting this parameter), indicating that you want a fixed number of send buffers. Getting the error indicates that you underestimated the number of buffers that you needed.

A send buffer cannot be reused until the data in it has been copied to the receiving PE. If the data is smaller than the size of a short message (`PVM_DATA_MAX`, which has a default of 4096 bytes), this copy occurs on the `pvm_send`(3) call. For larger amounts of data, this copy does not occur until the receiving PE has unpacked that data.

Make sure you are using buffers efficiently. Sometimes users convert code to use PVM, and the code appears as follows:

```
for (... several PEs ...) {
        pvm_initsend (PvmDataRaw);
        pvm_pkbyte (addr, size,...);
        pvm_send (...);
}
```

Here, the same data is being sent to each PE. However, a single packed buffer can be used by multiple sends:

```
pvm_initsend (PvmDataRaw);
pvm_pkbyte (addr, size, ...);
for (... several PEs ...) {
        pvm_send (...);
}
```

Or the single packed buffer can be used by a more efficient broadcast or multicast such as the following example:

```
pvm_initsend (PvmDataRaw);
pvm_pkbyte (addr, size, ...);
pvm_mcast (...);
```

In both cases, a single send buffer is used. The data it contains is not freed until all of the receiving PEs have responded, which may take a while; however, your use of buffers and memory will be reduced. Also, your program will run faster due to the reduced number of function calls.

3.4.3 `PvmOutOfResGmems`

PVM allows groups to consist of as many PEs as you specify, up to the total number of PEs in the partition. This is a general feature, but for large numbers of PEs it can waste memory. This is especially true if your groups are small relative to the number of PEs.

You can reduce the limit, and thus save memory, in either of two ways:

- Set the environment variable `PVM_MAXGTIDS`.
- Call `pvm_setopt`(3) with the `PvmMaxgtids` option (if this is done, the function must be called on each PE before any groups are formed).

Remember that the UNICOS/mk version of PVM defines a*global group*, consisting of all PEs in the partition. If you have code in which each PE is joining a global group with your own name (perhaps code ported from a network version of PVM), you should consider using the predefined global group on the UNICOS/mk system. This will simplify your code, and you will get better performance when using barriers across the group or broadcasts to the group.

3.5 Distributed Mode

The following sections discuss several issues specific to the distributed mode of the UNICOS/mk version. Using this mode requires that you use the PVM daemon. If you are not familiar with the use of the PVM daemon, you may want to read Section 2.3, page 6, before reading this section.

The following discussion assumes that the application you are running is using two partitions in the UNICOS/mk system. This assumption is made only for the sake of simplicity; your application can use other Silicon Graphics systems, or other systems connected to your network. Most of the same issues still apply.

3.5.1 Major Issues

The following sections discuss several key issues related to the distributed mode. The issues are as follows:

- PE communication
- UNICOS/mk executable files
- UNICOS/mk tasks

- Cross-system dynamic groups

3.5.1.1 PE Communication

The PVM daemon runs on the UNICOS/mk system. A PE on the UNICOS/mk system communicates with the daemon and with PVM tasks outside its own partition. In theory, any PE can do so. But UNICOS limits the number of open files per application and the number of open sockets in the system. So, if a UNICOS/mk application running on a large number of PEs were to set up communications for each PE, it may hit either or both of these limits.

Socket communications are very slow, especially compared to the speed of communications between PEs. Because much of socket communication is single-threaded in the PVM daemon, the performance cost goes up as more PEs try to communicate at the same time.

For these reasons, by default, only PE 0 establishes communications with the daemon, and you should consider using PVM in this manner. However, you can specify additional PEs by setting the `PVM_PE_LIST` environment variable, as follows:

```
setenv PVM_PE_LIST 0,4,8,12
setenv PVM_PE_LIST all
```

This environment variable must be set for both the PVM daemon `pvmd3`(1) and the application to read, and both must read the same value.

Note: At present, PE 0 always establishes communications with the daemon, even if PE 0 is not specified in `PVM_PE_LIST`. It is suggested that `PVM_PE_LIST` specify PE 0, if it is being used, to ensure future compatibility. It is possible that future releases may introduce other mechanisms for controlling access to the daemon.

3.5.1.2 UNICOS/mk Executable Files

When you build your UNICOS/mk executable file, you can optionally fix the number of PEs at load time. For such executable files, the `pvm_spawn`(3) *count* parameter simply specifies the size of the *tids* array, and must be at least as large as the PE count.

If you do not fix the number of PEs (for example, by using the `-Xm` option with `cld`(1)), you have a *malleable* executable file. For these, the `pvm_spawn` *count* parameter specifies the number of PEs that you want for the executable file.

When `pvm_spawn` returns successfully, it returns a count value that specifies the number of PEs that were started. The *tids* array is set with either of two values in each entry:

- For PEs that can communicate with the daemon, the associated entry contains a *pvm_tid* value.
- For PEs that cannot communicate with the daemon, the associated entry contains the integer value 1, which is not a valid *pvm_tid* value.

3.5.1.3 UNICOS/mk Tasks

During startup, the UNICOS/mk program checks to determine if the PVM daemon is running. If it is not, the program assumes it is in stand-alone mode.

3.5.1.4 Cross-system Dynamic Groups

You cannot form a dynamic group consisting of tasks from the UNICOS/mk system and another system. You cannot form a dynamic group consisting of tasks from more than one partition within a UNICOS/mk system. You must view group handling on each system and partition as being completely independent. If the UNICOS/mk tasks form a group called `MYGROUP`, and the tasks in the network also join a group called `MYGROUP`, the two groups are completely independent. A broadcast from a UNICOS/mk task to `MYGROUP` sends messages only within that partition; no messages will go outside the partition.

Note: In future releases, this limitation might be removed. Therefore, you should not build your application assuming that the two groups are independent; in a later release, they might form a single, combined group.

3.5.2 Session Example

You can use programs and commands a number of different ways to run a distributed application involving the UNICOS/mk system. The following example shows one way.

Example 3: Parent task spawning a child task

Assume that the parent task runs on a single PE in the UNICOS/mk system and uses PVM. The key line of interest is the call to `pvm_spawn`(3). There are several options for making this call. The following is a typical call:

```
count = pvm_spawn("mpp.a.out", 0, PvmTaskArch, "CRAY", nproc, tids);
```

In the example, a variable, *nproc*, specifies the size of the `tids` array. If the executable file (`mpp.a.out`) is built with a fixed PE count, *nproc* must be larger than or equal to the PE count, and `count` returns the PE count. If the executable file is built as a malleable executable file (that is, the number of PEs is not fixed), *nproc* is the number of PEs to request, and `count` returns the same number.

By specifying that the task should run on a Cray system, the code assumes that any Cray system in the virtual machine is acceptable. If not, `PvmTaskHost` should be specified instead of `PvmTaskArch`.

There is little out of the ordinary in the parent task. It must be careful not to use entries in the *pvm_tid* array that are set to a value of 1. It can communicate with any other PE assigned to the executable file.

The child task on the UNICOS/mk system does not look very different from one written to run in stand-alone mode. You must be careful to use *pvm_tid*, instead of PE numbers, when referring to the parent task. You must also be careful that only those PEs that can communicate with the daemon try to do so. You can deal with both of these constraints by calling `pvm_parent`(3). If this function returns a *pvm_tid*, that identifier can be used for communication. If the function returns the `PvmNoParent` error, that PE cannot communicate with the outside world. Section 2.3, page 6, describes how to start the PVM daemon and your parent task.

3.5.3 System Calls and PVM

In distributed mode, PVM uses sockets for communication. Read and write system calls actually transmit control and data across the sockets. Further, a given PVM task may have several sockets open at once: one to its local daemon and, optionally, one or more to specific tasks with which it is communicating.

The following facts have important implications regarding performance:

- System calls perform the I/O.
- There is a maximum size applied to data in a socket when it is transmitted or received; the system divides up requests larger than this maximum.
- With multiple open sockets, it is necessary to use yet another call, `select`(2), to look for incoming data or to determine if data can be output.

By default, in distributed mode, only PE 0 communicates with the PVM daemon, but additional PEs can also be permitted to communicate (for more information, see Section 3.5.1.1, page 59). If you are interested in performance, think very carefully before using more than one PE to make PVM calls outside the UNICOS/mk partition. This guideline applies regardless of the other options discussed in this chapter.

Because distributed mode is so dependent upon system calls, you should not use it for sending small, frequent messages.

3.5.4 Data Conversion

If you are using PVM to communicate between a UNICOS/mk system and a UNICOS system with Cray floating-point hardware, and you specify `PvmDataDefault` when calling `pvm_initsend`(3), PVM converts the data between IEEE and Cray formats for all forms of typed data. This is not done very efficiently on the UNICOS end.

You can perform data conversion efficiently on the UNICOS system, however, by using the data conversion functions available in the UNICOS Fortran libraries (see the *Application Programmer's Library Reference Manual*). If you are using PVM to transfer the data, pack and unpack it with the byte options (`pvm_pkbyte`(3), `pvm_upkbyte`(3), or the Fortran `BYTE1` option) and then call `CRAY2IEG`(3) or `IEG2CRAY`(3), as appropriate. If you are using file I/O, call `CRAY2IEG` or `IEG2CRAY`, as appropriate, on the data you are about to write from the UNICOS system or have just read from the UNICOS/mk system.

If you are using file I/O, an easier option is to use a Fortran I/O feature that automatically converts data as it is read or written. These techniques are described in the *Application Programmer's I/O Guide*.

Functions and Subroutines [4]

This chapter provides general information about PVM error messages and include files, and briefly describes tasks and associated functions.

You can use the C and Fortran interfaces to the PVM library functions to perform the following kinds of tasks:

- Basic operations (see Section 4.4, page 65)
- Task control (see Section 4.5, page 66)
- Option management (see Section 4.6, page 66)
- Dynamic system configuration (see Section 4.7, page 67)
- Dynamic task group management (see Section 4.8, page 67)
- Data transmittal (see Section 4.9, page 68)
- Data receipt (see Section 4.10, page 69)
- Barrier synchronization (see Section 4.11, page 71)
- Global operations (see Section 4.12, page 72)
- Signaling (see Section 4.13, page 73)
- Error handling (see Section 4.14, page 73)

This chapter briefly describes these tasks. The functions associated with each task are listed in a table. In each table, the functions are grouped as they are described on the man pages, and the groups are listed in the order you usually use them to perform the tasks.

In most cases, each logical PVM function is represented by a C function and a Fortran subroutine. For more information about a specific function or subroutine, use the `man`(1) command to view the associated man page online. To simplify references, this discussion refers to C functions, C++ functions, and Fortran subroutines as *functions* unless individual differences require documentation.

When the C interfaces specify `char *` as a data type, the Fortran interfaces generally permit specification of Fortran character variables or constants. However, these Fortran values are processed as C strings; therefore, a null character in the middle of the character sequence, which is valid in Fortran, terminates the string.

4.1 Error Messages

For a complete list of the PVM error messages and the value associated with each, see Appendix A, page 75. In general, PVM functions return `PvmOk` (0) or a negative number for errors. Some functions return positive values with other meanings or have special return codes. Error checks should be coded as less than 0, rather than not equal to 0.

You can control the actions that PVM takes when it detects an error. The default is to print an ASCII message and return an error code to the caller. For more information, see the `pvm_setopt`(3) man page for a description of the `PvmAutoErr` option.

4.2 Process Identifiers

All processes that enroll in PVM are represented by an integer task identifier, a `pvm_tid`. Because `pvm_tid` values must be unique across the entire virtual machine, they are supplied by PVM and are not chosen by the user. The following routines return `pvm_tid` values:

`pvm_bufinfo`(3)

`pvm_gettid`(3)

`pvm_mytid`(3)

`pvm_parent`(3)

`pvm_spawn`(3)

4.3 PVM Include Files

PVM include files for the MPT release are installed in the `$PVM_ROOT/include` directory. If the `mpt` module has been loaded, this include file directory will be searched before any standard include directories.

For better portability, you can refer to PVM include files in your source and specify the include file directory on the compiler command line, as follows:

From C:

```
#include <pvm3.h>
cc -I $PVM_ROOT/include
```

From Fortran:

```
include "fpvm3.h"
f90 -I $PVM_ROOT/include
```

Note: PVM include files may exist in the `/usr/include` directory if your site has also installed the Cray network version of PVM. Be careful not to use those files by mistake.

4.4 Basic Operations

You can perform basic PVM operations by using the functions in Table 14. Some of the functions are standard PVM shared memory implementation features for UNICOS/mk systems, but represent an implementation extension for UNICOS systems. These are marked “UNICOS extension.”

Table 14. Basic Operations Functions

C and C++ function	Fortran subroutine	Description
`_my_pe`	`MY_PE`	Returns the PE number of the PVM task that calls it (UNICOS extension)
`_num_pes`	`NUM_PES`	Returns the total number of PEs (or PVM tasks) in the program (UNICOS extension)
`pvm_freezegroup`	`PVMFFREEZEGROUP`	Freezes dynamic group membership and caches information locally
`pvm_get_PE`	`PVMFGETPE`	Converts a task ID into a PE number (UNICOS extension)
`pvm_hostsync`	`PVMFHOSTSYNC`	Gets the time-of-day clock from the PVM host
`pvm_mytid`	`PVMFMYTID`	Returns the `pvm_tid` of the calling task

C and C++ function	Fortran subroutine	Description
`pvm_parent`	`PVMFPARENT`	Returns the `pvm_tid` for the task that spawned the calling task
`pvm_tidtohost`	`PVMFTIDTOHOST`	Returns the `pvm_tid` for the PVM daemon task

4.5 Task Control

You can control PVM process creation and termination by using the task control functions in Table 15.

Table 15. Task Control Functions

C and C++ function	Fortran subroutine	Description
`pvm_catchout`	`PVMFCATCHOUT`	Catches output from child tasks
`pvm_exit`	`PVMFEXIT`	Exits PVM
`pvm_halt`	`PVMFHALT`	Shuts down the entire PVM system
`pvm_kill`	`PVMFKILL`	Terminates a PVM task
`pvm_pstat`	`PVMFPSTAT`	Determines if a PVM task is executing
`pvm_reg_hoster`	(Not applicable)	Registers a task as the PVM host starter
`pvm_reg_tasker`	(Not applicable)	Registers a task as the PVM task starter
`pvm_spawn`	`PVMFSPAWN`	Starts a new PVM task

4.6 Option Management

You can control PVM options by using the functions in Table 16.

Table 16. Option Management Functions

C and C++ function	Fortran subroutine	Description
`pvm_setopt`	`PVMFSETOPT`	Sets a PVM option
`pvm_getopt`	`PVMFGETOPT`	Returns the current value of a PVM option

4.7 Dynamic System Configuration

The dynamic system configuration functions, described in Table 17, allow PVM to be dynamically configured by the application. Systems may be added or removed from the virtual machine, and information can be obtained about a particular system or about the virtual machine as a whole.

Table 17. Dynamic System Configuration Functions

C and C++ function	Fortran subroutine	Description
`pvm_addhosts` `pvm_delhosts`	`PVMFADDHOST` `PVMFDELHOST`	Adds or deletes one or more systems
`pvm_config`	`PVMFCONFIG`	Returns the configuration of the virtual machine
`pvm_mstat`	`PVMFMSTAT`	Returns the status of the specified system
`pvm_tasks`	`PVMFTASKS`	Returns information about tasks

4.8 Dynamic Task Group Management

A PVM application can form dynamic groups of tasks during its execution. Usually, these groups are established to simplify *multicasting* (the broadcast of data to a number of tasks) and barrier synchronization. Tasks can join and leave groups as desired.

A group is identified by a character string that is assigned by the user. All tasks that want to join a group must specify the same character string.

Dynamically joining and leaving a group must be done with care. Synchronization problems can arise if, for example, one task is joining a group at the same time another task is broadcasting a message to the group. Participating tasks should synchronize at a barrier before trying to use a group.

Dynamic task group management functions are described in Table 18.

Table 18. Dynamic Task Group Management Functions

C and C++ function	Fortran subroutine	Description
`pvm_getinst`	`PVMFGETINST`	Returns the instance number of a task
`pvm_gettid`	`PVMFGETTID`	Returns the `pvm_tid` for a task
`pvm_gsize`	`PVMFGSIZE`	Returns the number of tasks in a group
`pvm_joingroup` `pvm_lvgroup`	`PVMFJOINGROUP` `PVMFLVGROUP`	Joins or leaves a dynamic group

4.9 Data Transmittal

There are two methods in PVM for sending messages. The simpler method, which involves the use of the `pvm_psend`(3) function, lets you make a single call to transmit a contiguous block of data to another PVM task.

The more complex method involves three steps:

1. Initializing a send buffer
2. Packing one or more blocks of data into the buffer
3. Transmitting the buffer to one or more tasks

The second method is more powerful and flexible than the first, but runs more slowly. Messages can be sent to a particular task, can be broadcast to all members of a group, can be broadcast to all tasks, or can be multicast to a list of tasks.

You can use the data transmittal functions in Table 19, to transmit data.

Table 19. Data Transmittal Functions

C and C++ function	Fortran subroutine	Description
`pvm_bcast`	`PVMFBCAST`	Broadcasts a message to all tasks in a group.
`pvm_getsbuf`	`PVMFGETSBUF`	Returns the buffer identifier of the current send buffer.
`pvm_initsend`	`PVMFINITSEND`	Initializes a send buffer.
`pvm_mcast`	`PVMFMCAST`	Broadcasts a message to all tasks in an array.
`pvm_mkbuf` `pvm_freebuf`	`PVMFMKBUF` `PVMFFREEBUF`	Creates send buffers or releases buffers.
`pvm_psend`	`PVMFPSEND`	Packs and sends data in one call.

`pvm_pkint`
`pvm_pkshort`
`pvm_pklong`
`pvm_pkuint`
`pvm_pkushort`
`pvm_pkulong`
`pvm_pkfloat`
`pvm_pkdouble`
`pvm_pkcplx`
`pvm_pkdcplx`
`pvm_pkbyte`
`pvm_pkstr`
`pvm_packf`
	`PVMFPACK`	Inserts data values into the send buffer. See `pvm_pk`(3).

`pvm_send`	`PVMFSEND`	Sends a message to a single task.
`pvm_setsbuf`	`PVMFSETSBUF`	Specifies a new buffer as the current send buffer.

4.10 Data Receipt

There are two methods in PVM for receiving messages. The simpler method, which involves the use of the `pvm_precv`(3) function, lets you make a single call to receive a message and store its data into a contiguous block of data. This is a *blocking receive*; the calling task does not return until an appropriate message arrives.

The more complex method involves two steps:

1. Receiving a message. (You can choose either a blocking or a nonblocking form of receive.)
2. Unpacking one or more blocks of data from the message.

Both methods allow you to choose the message to receive. You can choose to receive a message of any of the following types:

- A message with a specific message tag sent by a specific PVM task
- Any message sent by a specific PVM task
- A message with a specific message tag sent by any PVM task
- Any message at all

In addition, PVM provides an optional capability that lets you select a message based on any criteria (including the contents of the message itself). To use this feature, you must write a comparison function (in C) and call `pvm_recvf`(3) or `pvm_trecv`(3). PVM then calls this comparison function on each subsequent `pvm_recv`(3) or `pvm_nrecv`(3) call to identify the message that should be selected.

After a message has been received, the data is available in an internal receive buffer, and additional functions must be called to transfer (and convert) this data into user buffers. Any combination and number of calls to the unpacking functions may be made to move this data into user memory, but it is recommended that the sequence of unpacking calls match the sequence of packing calls that built up the data for the message. It may be possible to use a different sequence, but you should be aware that this depends on undocumented, underlying data packing and transfer mechanisms. (This is particularly dangerous if you use `pvm_pkstr`(3) or if you use `pvm_pkbyte`(3) with a byte count that is not a multiple of 8. Also, if you ever anticipate using this code on another system or across heterogeneous systems, you should avoid using a different sequence.)

The data receipt functions are described in Table 20.

Table 20. Data Receipt Functions

C and C++ function	Fortran subroutine	Description
`pvm_bufinfo`	`PVMFBUFINFO`	Returns information about a message.
`pvm_freebuf`	`PVMFFREEBUF`	Releases receive buffers. See `pvm_mkbuf`(3).
`pvm_getrbuf`	`PVMFGETRBUF`	Returns the buffer identifier of the current receive buffer.
`pvm_precv`	`PVMFPRECV`	Receives a message directly into a buffer.
`pvm_recv` `pvm_nrecv` `pvm_probe`	`PVMFRECV` `PVMFNRECV` `PVMFPROBE`	Receives a message or probes for a message.
`pvm_recvf`	(Not applicable)	Supplies a user-written comparison function.
`pvm_setrbuf`	`PVMFSETRBUF`	Specifies a new buffer as the current receive buffer.
`pvm_trecv`	`PVMFTRECV`	Receives a message with a time-out.
`pvm_upkint` `pvm_upkshort` `pvm_upklong` `pvm_upkuint` `pvm_upkushort` `pvm_upkulong` `pvm_upkfloat` `pvm_upkdouble` `pvm_upkcplx` `pvm_upkdcplx` `pvm_upkbyte` `pvm_upkstr` `pvm_unpackf`	`PVMFUNPACK`	Extracts values from received messages. See `pvm_upk`(3).

4.11 Barrier Synchronization

The `pvm_barrier`(3) function described in Table 21 lets PVM tasks explicitly synchronize with one another. Calling this function causes the task to *block* (wait) until a specified number of tasks in a group have called the function.

When this occurs, all waiting tasks are unblocked. The calling task must be a member of the group, and the *count* argument must be the same for all tasks that use the same barrier.

The `barrier`(3) function described in Table 21 lets multitasked PVM tasks explicitly synchronize with one another. This function is useful when PVM is being used in stand-alone mode for global synchronization between all multitasked PVM tasks.

Table 21. Barrier Synchronization Function

C and C++ function	Fortran subroutine	Description
`barrier`	`BARRIER`	Creates a barrier to synchronize multitasked PVM tasks (UNICOS/mk and UNICOS PVM shared memory implementation extension)
`pvm_barrier`	`PVMFBARRIER`	Creates a barrier to synchronize tasks

4.12 Global Operations

The functions in Table 22 allow the tasks in a group to participate in a global operation. All tasks in the group must call the same function at the same time.

The `pvm_reduce`(3) function supports sum, product, max, and min operations, as well as user-defined operations.

Table 22. Global Operations Functions

C and C++ function	Fortran subroutine	Description
`pvm_gather`	`PVMFGATHER`	Gathers data from group members into an array
`pvm_reduce`	`PVMFREDUCE`	Performs a reduction operation across a group
`pvm_scatter`	`PVMFSCATTER`	Sends a section of an array to each member of the group

4.13 Signaling

The functions in Table 23 support sending signals of different kinds to PVM tasks.

Table 23. Signaling Functions

C and C++ function	Fortran subroutine	Description
`pvm_notify`	`PVMFNOTIFY`	Notifies tasks of specific events
`pvm_sendsig`	`PVMFSENDSIG`	Sends a signal to a task

4.14 Error Handling

The function in Table 24 provides simple help for handling PVM-generated errors.

Table 24. Error Handling Function

C and C++ function	Fortran subroutine	Description
`pvm_perror`	`PVMFPERROR`	Outputs a PVM error message

For more information on controlling PVM behavior, see the `pvm_setopt`(3) man page.

PVM Error Messages [A]

Table 25 lists the errors detected by PVM. These error message descriptions include the following information:

- Text of the error message written to standard error by PVM functions
- Numeric value of the error returned by PVM functions
- Symbol name for each error, as defined within the PVM include files
- Additional information about the error

Be cautious in your use of the numeric values, because the values assigned to the symbols may change at any time and without any notice.

Errors with numeric values of –100 and below are Silicon Graphics extensions.

Table 25. Error Messages Issued by PVM Functions

Error text	Value	Symbol	Additional information
	0	`PvmOk`	
	–1		Reserved
`Bad parameter`	–2	`PvmBadParam`	A bad parameter was passed to the function.
`Count mismatch`	–3	`PvmMismatch`	The count parameter does not match the count used in peer tasks.
`Value too large`	–4	`PvmOverflow`	A value is too large to be packed or unpacked.
`End of buffer`	–5	`PvmNoData`	The end of a message buffer was reached while trying to unpack data.
`No such host`	–6	`PvmNoHost`	There is no host in the virtual machine with the specified name, or the name could not be resolved to an address.
`No such file`	–7	`PvmNoFile`	The specified executable file does not exist.
	–8		Reserved

Error text	Value	Symbol	Additional information
	–9		Reserved
`Malloc failed`	–10	`PvmNoMem`	`malloc` failed to get memory for `libpvm`.
	–11		Reserved
`Can't decode message`	–12	`PvmBadMsg`	The received message has a data format native to another machine, which cannot be decoded by `libpvm`.
	–13		Reserved
`System error`	–14	`PvmSysErr`	`libpvm` could not contact a `pvmd` daemon on the local host, or the `pvmd` failed during an operation.
`No current buffer`	–15	`PvmNoBuf`	There is no current message buffer to pack or unpack.
`No such buffer`	–16	`PvmNoSuchBuf`	There is no message buffer with the specified buffer handle.
`Null group name`	–17	`PvmNullGroup`	A null group name was passed to a function.
`Already in group`	–18	`PvmDupGroup`	The task is already a member of the group it attempted to join.
`No such group`	–19	`PvmNoGroup`	The specified group does not exist.
`Not in group`	–20	`PvmNotInGroup`	The specified group has no such member task.
`No such instance`	–21	`PvmNoInst`	The specified group has no member with this instance.
`Host failed`	–22	`PvmHostFail`	A foreign host in the virtual machine failed during the requested operation.
`No parent task`	–23	`PvmNoParent`	This task has no parent task.
`Not implemented`	–24	`PvmNotImpl`	This `libpvm` function or option is not implemented.
`Pvmd system error`	–25	`PvmDSysErr`	An internal mechanism in the `pvmd` daemon failed during the requested operation.

Error text	Value	Symbol	Additional information
`Version mismatch`	–26	`PvmBadVersion`	Two PVM components (a `pvmd` daemon and a task, two `pvmd` daemons, or two tasks) have incompatible protocol versions and cannot interoperate.
`Out of resources`	–27	`PvmOutofRes`	The requested operation could not be completed due to lack of resources.
`Duplicate host`	–28	`PvmDupHost`	An attempt was made to add the same host to a virtual machine more than once, or to add a host already a member of another virtual machine owned by the same user.
`Can't start pvmd`	–29	`PvmCantStart`	A `pvmd` daemon could not be started on the local host, or a slave `pvmd` daemon could not be started on a remote host.
`Already in progress`	–30	`PvmAlready`	The requested operation requires exclusive access, and another operation was already in progress.
`No such task`	–31	`PvmNoTask`	No task exists with the given TID.
`No such entry`	–32	`PvmNoEntry`	The class server has no entry matching the lookup request.
`Duplicate entry`	–33	`PvmDupEntry`	The class server already has an entry matching the insert request.
`Name too long`	–100	`PvmTooLong`	
`Async transfers still active`	–101	`PvmStillActive`	
`Precision lost on default pack`	–102	`PvmLostPrecision`	
`Out of buffers`	–103	`PvmOutOfResBuf`	The requested operation could not be completed due to lack of data buffer resources.
`Out of shared memory pool`	–104	`PvmOutOfResSMP`	The requested operation could not be completed due to lack of SMP resources.

Error text	Value	Symbol	Additional information
`Too many group members`	–105	`PvmOutOfResGmems`	The requested operation could not be completed due to lack of resources.
`Too much data packed`	–106	`PvmTooMuchData`	
`Hit PVM_TOTAL_PACK limit`	–107	`PvmMemLimit`	
`Cannot communicate`	–200	`PvmNoCom`	A multitasked task cannot communicate with the PVM daemon.

PVM Man Pages [B]

The following list shows the online PVM man pages, which document the specified commands and functions (arranged alphabetically).

`man1` pages:

- `pvm_intro`(1)
- `pvm`(1)
- `pvmd3`(1)

`man3` pages:

- `pvm_addhosts`(3)
- `pvm_barrier`(3)
- `pvm_bcast`(3)
- `pvm_bufinfo`(3)
- `pvm_catchout`(3)
- `pvm_channels`(3)
- `pvm_config`(3)
- `pvm_disptrace`(3)
- `pvm_exit`(3)
- `pvm_freezegroup`(3)
- `pvm_gather`(3)
- `pvm_getfds`(3)
- `pvm_get_PE`(3)
- `pvm_getinst`(3)
- `pvm_getrbuf`(3)
- `pvm_getsbuf`(3)
- `pvm_gettid`(3)

- `pvm_gsize`(3)
- `pvm_halt`(3)
- `pvm_hostsync`(3)
- `pvm_initsend`(3)
- `pvm_joingroup`(3)
- `pvm_kill`(3)
- `pvm_mcast`(3)
- `pvm_mkbuf`(3)
- `pvm_mstat`(3)
- `pvm_mytid`(3)
- `pvm_notify`(3)
- `pvm_parent`(3)
- `pvm_perror`(3)
- `pvm_pk`(3)
- `pvm_precv`(3)
- `pvm_psend`(3)
- `pvm_pstat`(3)
- `pvm_recv`(3)
- `pvm_recvf`(3)
- `pvm_reduce`(3)
- `pvm_reg_hoster`(3)
- `pvm_reg_tasker`(3)
- `pvm_scatter`(3)
- `pvm_send`(3)
- `pvm_sendsig`(3)
- `pvm_setopt`(3)

- `pvm_setrbuf`(3)
- `pvm_setsbuf`(3)
- `pvm_spawn`(3)
- `pvm_tasks`(3)
- `pvm_tidtohost`(3)
- `pvm_trecv`(3)
- `pvm_upk`(3)

Glossary

asynchronous

An asynchronous operation or function proceeds in parallel with its initiator. The initiator must check later to see if the operation or function has completed.

blocking

A blocking function is one that does not return until the function is complete.

broadcast

To send messages to multiple tasks. Often, a *broadcast* is used in the sense of sending to all tasks, whereas *multicast* is used in the sense of sending to an arbitrary set of tasks.

`cplx`

A data item consisting of two successive `float` types.

`dcplx`

A data item consisting of two successive `double` types.

dynamic groups

Groups in which tasks can join and leave groups at any time.

EU

Emory University.

global groups

A group consisting of all the tasks (or PEs) in the MPP partition.

message passing

A parallel programming style in which explicit messages (containing a user-defined, integer message type and data) are sent between tasks.

multicast

To send messages to multiple tasks. See also *broadcast*.

nonblocking

A nonblocking function is one that returns immediately.

NQE

Network Queuing Environment.

ORNL

Oak Ridge National Laboratory.

partition

A collection of PEs that are assigned to a job running on Cray MPP systems.

PE

Processing element.

probe

A message passing concept in which a check is made to see if a message is available, though the message is not actually received at that time.

PVM

Parallel Virtual Machine.

PVM console

A user-level command that lets you monitor and control your PVM system. The console is run with the command `pvm`.

PVM daemon

A user-level process that controls and manages PVM activity on a given host machine. The daemon is run with the command `pvmd3`.

pvm_tid

The name used in this manual to refer to a PVM task identifier, which is used to reference a specific PVM task.

RPC

Remote Procedure Call.

SIMD

Single instruction, multiple data.

SPMD

Same program, multiple data.

Stand-alone mode

PVM is used for communication between tasks within a single executable file with no PVM daemon present. On Cray PVP machines, this mode uses the Cray multitasking function to provide memory as a PVM task communication mechanism, which offers enhanced communication.

stride

The spacing between elements.

synchronous

A synchronous operation or function does not return control to its initiator until it has completed the requested operation or function.

task

An independent, parallel process.

task identifier

A 32-bit integer uniquely identifying a PVM task.

UDP

User datagram protocol.

UT
University of Tennessee.

XDR
eXternal Data Representation.

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3686-002.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389