PVM and HeNCE Programmer's Manual

SR–2501 5.0

Cray Research, Inc.

PVM (Parallel Virtual Machine) and HeNCE (Heterogeneous Network Computing Environment) were developed by the Oak Ridge National Laboratory, the University of Tennessee, and Emory University. This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy , in part by the National Science Foundation, and in part by the State of Tennessee.

The UNICOS operating system is derived from the UNIX System Laboratories, Inc. UNIX System V operating system. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

Requests for copies of Cray Research, Inc. publications should be sent to the following address:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120
USA

Order desk (612) 683–5907
Fax number (612) 452–0141

New Features

PVM and HeNCE Programmer's Manual SR–2501 5.0

This update of the *PVM and HeNCE Programmer's Manual* supports two new releases: Cray Network PVM-3, version 2.1 and Cray MPP PVM-3, version 1.2. The following new features have been added:

- Nine standard PVM functions
  - `pvm_catchout`(3)
  - `pvm_gather`(3)
  - `pvm_halt`(3)
  - `pvm_hostsync`(3)
  - `pvm_reduce`(3)
  - `pvm_reg_hoster`(3)
  - `pvm_reg_tasker`(3)
  - `pvm_scatter`(3)
  - `pvm_trecv`(3)
- `htool`(1) command for HeNCE
- `PVM_RETRY_COUNT` environment variable and `pvm_setopt`(3) option

The following standard PVM functions are now documented on separate man pages:

- `pvm_bcast`(3)
- `pvm_getrbuf`(3)
- `pvm_getsbuf`(3)
- `pvm_mcast`(3)
- `pvm_send`(3)
- `pvm_setrbuf`(3)
- `pvm_setsbuf`(3)

The HeNCE sections have been updated, the set-up instructions have been rewritten, and a program example has been replaced.

The following incompatibilities from previous releases have been introduced:

- To avoid name conflicts, two C structure names have been changed. A new field, `ti_pid`, has also been added to the `pvmtaskinfo` structure. If your C programs reference the following structures, change the declarations; your Fortran programs are not affected:
  - `hostinfo` is now `pvmhostinfo`.
  - `taskinfo` is now `pvmtaskinfo`.

- In the network version of PVM, the following *hostfile* options for the `pvmd3`(1) command have changed:
  - `ms` is now `so=ms`.
  - `pw` is now `so=pw`.
- The PVM architecture name for Silicon Graphics IRIS systems using OS releases 5.0 or later is now `SGI5`.

Record of Revision

The date of printing or software version number is indicated in the footer. Changes in rewrites are noted by revision bars along the margin of the page.

Version	***Description***
1.0	August 1992. Original printing. This manual describes the use of PVM and HeNCE as released by Cray Research, Inc.
1.1	August 1992. Revision to support the release of the CRAY T3D emulator . Sections 4 and 5 have been added, along with an appendix contrasting the differences between PVM version 2.x and PVM version 3.0.
2.0	May 1993. Revision to support version 3.0 of PVM on the CRAY T3D emulator . Section 5 has been updated to support the new function calls, and appendix B has been added to give a complete list of the PVM error messages. Due to extensive changes in this rewrite, change bars were not used.
3.0	November 1993. Rewrite to support the following Cray releases: • Cray Network PVM-3 version 2.0 • Cray MPP PVM-3 version 1.0 • CRAY T3D Emulator 1.01 The manual has been reorganized and substantially rewritten since the last release. Due to extensive changes in this rewrite, change bars are not included.
4.0	June 1994. Rewrite to support the following Cray releases: • Cray MPP PVM-3, version 1.1 • Cray Network PVM-3, version 2.0 • CRAY T3D Emulator, release 1.0 or later Revision 4.0 applies to the manual, not the releases. The manual has been reorganized and substantially rewritten since the last release. Due to extensive changes in this rewrite, change bars are not included.

Version	*Description*
5.0	October 1994. Revision to support the following Cray releases: • Cray MPP PVM-3, version 1.2 • Cray Network PVM-3, version 2.1 • CRAY T3D Emulator, release 1.0 or later Revision 5.0 applies to the manual, not the releases.

Preface

The *PVM and HeNCE Programmer's Manual* describes the use of these products in a variety of computing environments, including Cray Research massively parallel processing (MPP) systems and the emulator. It is designed to function as a single source of information for these products during the programming, development, and use in a massively parallel processing environment.

This manual is divided into two parts: the first describes the Parallel Virtual Machine (PVM), and the second describes the Heterogeneous Network Computing Environment (HeNCE).

Acknowledgments

The PVM and HeNCE projects are being carried out by researchers at the Oak Ridge National Laboratory (ORNL), the University of Tennessee (UT), and Emory University (EU), with assistance from people from other universities and locations. The key researchers include the following:

- Adam Beguelin (Carnegie Mellon University)
- Jack Dongarra (ORNL & UT)
- G. A. Geist (ORNL)
- Weicheng Jiang (UT)
- Robert Manchek (UT)
- Vaidy Sunderam (EU)

Most of the material in this manual was taken from documentation written by this group.

Related publications

The following documents contain additional information that may be helpful:

- *Application Programmer's I/O Guide*, publication SG–2168

- *CRAY T3D Emulator User's Guide*, publication SG–2500
- *Cray Research MPP Software Guide*, publication SG–2508
- *I/O User's Guide*, publication SG–3075
- *Advanced I/O User's Guide*, publication SG–3076
- *UNICOS System Calls Reference Manual*, publication SR–2012
- *Cray Standard C Reference Manual*, publication SR–2074
- *UNICOS System Libraries Reference Manual*, publication SR–2080
- *Application Programmer Library Reference Manual*, publication SR–2165
- *Cray C++ Compiling System Technical Note*, publication SN–2131
- *Cray MPP Fortran Reference Manual*, publication SR–2504
- *Cray Standard C Reference Manual for MPP*, publication SR–2506
- *Parallel Virtual Machine (PVM) Reference Card*, publication SQ–2512
- *SHMEM Technical Note for Fortran*, publication SN–2516
- *SHMEM Technical Note for C*, publication SN–2517

The *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software manuals that are available to customers.

To order a manual, either call the Distribution Center in Mendota Heights, Minnesota, at (612) 683–5907 or send a facsimile of your request to fax number (612) 452–0141. Cray Research employees may send electronic mail to `orderdsk` (UNIX system users).

Conventions

The following conventions are used throughout this manual:

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
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 line.
$\boxed{\texttt{KEY}}$	This convention indicates a key on the keyboard.

The following machine naming conventions are used throughout this manual:

Term	Definition
CRAY Y-MP systems	All configurations of CRAY Y-MP systems supported by UNICOS 8.0, including the M90 series (M92, M94, M98); C90 series (C916, C92A, C94, C94A, and C98); E series (2E, 4E, 8E, and 8I); EL series (including CRAY Y-MP EL, CRAY EL92, and CRAY EL98).
CRAY X-MP systems	All configurations of CRAY X-MP systems supported by UNICOS 8.0. This includes CRAY X-MP, CRAY X-MP EA, and CRAY X-MP EMA systems.
CRAY-2 systems	All configurations of CRAY-2 systems supported by UNICOS 8.0.
Cray MPP systems	All configurations of the CRAY T3D series, supported by UNICOS 8.0, including CRAY T3D MC, CRAY T3D MCA, and CRAY T3D SC.
All Cray Research systems	All configurations supported by UNICOS 8.0.

In most cases, each logical PVM function is represented by a C and C++ function and a Fortran subroutine. To simplify references in this manual, *function* refers to both C and C++ functions and Fortran subroutines. For information about tasks that you can perform with these functions, see section 5. For syntax information, see the PVM man pages in appendix D or use the `man` command to view the man pages online. For a specific subroutine description, see the man page for the associated C and C++ function.

It is the objective of Cray Research to become compliant with IEEE Std 1003.1–1990 (POSIX.1) and IEEE Std 1003.2–1992 (POSIX.2). This manual reflects those ongoing efforts.

POSIX.2 uses *utility* to refer to executable programs that Cray Research documentation usually refers to as *commands* . Both terms appear in this document.

In this publication, *Cray Research*, *CRI*, and *Cray* refer to Cray Research, Inc. and/or its products.

Online information

The following types of online information products are available to Cray Research customers:

- CrayDoc online documentation reader, which lets you see the text and graphics of a manual online. The CrayDoc reader is available on workstations. To start the CrayDoc reader at your workstation, use the `cdoc`(1) command.
- Docview text-viewer system, which lets you see the text of a manual online. The Docview system is available on the Cray Research mainframe. To start the Docview system, use the `docview`(1) command.
- Man pages, which describe a particular element of the UNICOS operating system or a compatible product. To see a detailed description of a particular command or routine, use the `man`(1) command.
- UNICOS message system, which provides explanations of error messages. To see an explanation of a message, use the `explain`(1) command.
- Cray Research online glossary, which explains the terms used in a manual. To get a definition, use the `define`(1) command.
- `xhelp` help facility. This online help system is available within tools such as the Program Browser (`xbrowse`) and the MPP Apprentice tool.

For detailed information on these topics, see the *User's Guide to Online Information*, publication SG–2143.

Reader comments

If you have comments about the technical accuracy, content, or organization of this manual, please tell us. You can contact us in any of the following ways:

- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  `uunet!cray!publications`

- Send us electronic mail from any system connected to Internet, using the following Internet addresses:

  `pubs2501@timbuk.cray.com` (comments on this manual)

  `publications@timbuk.cray.com` (general comments)

- Contact your Cray Research representative and ask that a Software Problem Report (SPR) be filed. Use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.

- Call our Software Information Services department in Eagan, Minnesota, through the Technical Support Center, using either of the following numbers:

  (800) 950–2729 (toll free from the United States and Canada)

  (612) 683–5600

- Send a facsimile of your comments to the attention of "Software Information Services" in Eagan, Minnesota, at fax number (612) 683–5599.

- Use the postage-paid Reader's Comment Form at the back of the printed manual.

We value your comments and will respond to them promptly.

Contents

Figures

Tables

Part 1: PVM

Part 1 describes the Parallel Virtual Machine (PVM) as used on Cray Research systems. Part 1 is organized as follows:

Section	Title and description
1	*PVM Overview* contains general product information.
2	*Cray MPP Version* describes using PVM on Cray Research massively parallel processing (MPP) systems.
3	*Network Version* describes using PVM between computer systems on a network.
4	*Emulator Version* describes using PVM on the CRAY T3D emulator.
5	*Functions and Subroutines* describes the tasks you can perform with the C and C++ functions and Fortran subroutines.

Part 2 describes the Heterogeneous Network Computing Environment (HeNCE), a graphic interface for prototyping PVM programs.

The following appendixes provide additonal PVM information.

Section	Title and description
A	*PVM Error Messages* lists the PVM messages.
B	*Comparison of Message-passing Libraries* compares PVM on Cray MPP systems with the message-passing libraries supplied on other MPP systems.
C	*Differences between PVM Version 2 and Version 3* describes the PVM version 2 interface to aid in porting to PVM version 3.
D	*PVM Man Pages* contains copies of the PVM man pages.

For quick-reference PVM information, see the *Parallel Virtual Machine (PVM) Reference Card*, publication SQ–2512.

Section	Title and description

PVM Overview [1]

PVM (Parallel Virtual Machine) was initially developed as a software system to enable a collection of heterogeneous computer systems to be used as a coherent and flexible concurrent computation resource. Cray Research has taken this initial implementation and has extended it in several ways. This manual describes how this single product can be used to meet a variety of needs.

At its base, PVM is a subroutine library that supports parallel programming through a technique known as *message-passing*. This style of parallel programming is an explicit method in which the application specifically requests that data be sent from one task to another, or between groups of tasks. The PVM library also supports barrier synchronization.

The PVM product was developed to support parallel applications across a network of heterogeneous computer systems, but message-passing can also be used to program MPP systems. Most MPP systems today offer the user one or more message-passing libraries. Many of these libraries offer a vendor-specific interface, however, and an application programmed using one of these interfaces cannot easily be ported to another MPP system or to a networked cluster or system of other computers.

CRAY T3D systems are installed and are closely coupled with a CRAY C90 or CRAY Y-MP system. This is a heterogeneous system; the MPP system (the CRAY T3D computer) and the parallel vector system (the CRAY Y-MP computer) use different representations of floating-point data.

Cray Research provides versions of PVM to support a variety of needs. These versions provide users with a single, portable, de facto standard subroutine interface to do message-passing programming. PVM is available as public domain software from its developers and is being made available as vendor-supported software by Cray Research and a number of other computer vendors. By using PVM in your application, you will not be locked into a proprietary interface.

Cray Research PVM releases
1.1

Cray Research offers three release packages that include PVM libraries. All three versions are documented in this manual. Each version of PVM released by Cray Research has been developed for the systems on which it runs. Internally each is quite different, but to the application, the versions have much in common.

The three releases are as follows:

- Cray MPP PVM-3, version 1.2, subsequently referred to as the *Cray MPP version* (or the *CRAY T3D version*).
- Cray Network PVM-3, version 2.1, often referred to as the *network version*.
- CRAY T3D Emulator, release 1.0 or later, often referred to as the *emulator version*.

Note

Because of different release schedules, the precise set of functions supported in the three releases may differ at any given time. A function or option may appear in one release many months before it appears in another. Such discrepancies are documented in the PVM man pages, which you can reference in appendix D and online by using the `man`(1) command.

In particular, no new features are being added to the CRAY T3D emulator by Cray Research. As a result, newer PVM functions implemented in the other two versions are not being added to the emulator version.

Cray MPP version
1.1.1

PVM is supported by Cray Research as the standard message-passing library for Cray MPP systems. Cray MPP PVM-3, version 1.2, cannot be ordered separately; it is included as part of the CF77 Programming Environment 1.2 for MPP and the Cray Standard C Programming Environment 1.2 for MPP. For a discussion of specific issues related to this release, see section 2, page 9.

The Cray MPP version can be used in two modes:

- In stand-alone mode, it can be used to program an application on the CRAY T3D system, like the message-passing libraries supplied for other MPP systems.
- In distributed mode, it can be coupled with the network version to let CRAY T3D applications use message-passing to communicate with processes running on the CRAY Y-MP system or any of the other systems that run PVM.

The Cray MPP version uses the hardware capabilities of the CRAY T3D system to handle communications between CRAY T3D processor elements (PEs), and uses Transmission Control Protocol/Internet Protocol (TCP/IP) and the user datagram protocol (UDP) data transfers to handle communications outside the Cray MPP system. Between PEs, communications are fast, allowing the application more flexibility in determining how often to send messages and how much data to send per message.

Cray Research provides several extensions that allow a programmer to make trade-offs between portability and maximum performance. Very portable functions are provided and offer good performance; the Cray MPP version also provides functions that are not offered in the network version but can provide extremely good performance for some applications.

Network version 1.1.2

The Cray Network PVM-3, version 2.1, referred to here as the *network version*, is basically the same as the software developed and released by the PVM developers at Oak Ridge National Laboratory (ORNL) and elsewhere. This version supports PVM across a heterogeneous network of computers. For a discussion of specific issues related to this release, see section 3, page 41. (This is an asynchronous release that you order separately.)

The network version consists of a PVM library and several commands that support PVM. The most important of these is a user-level daemon, which runs on each computer system in the PVM system.

The network version is based on data transfers over UNIX sockets between UNIX processes. In this kind of communication, the network transfer speeds are relatively slow, and this means that applications using PVM should avoid sending many, small messages.

Emulator version 1.1.3

To assist in programming the CRAY T3D system, Cray Research supplies an emulator that runs on CRAY Y-MP systems. The emulator allows programs to be developed and executed without having to use a CRAY T3D system. PVM is included in the emulator library. The *emulator version* of PVM is part of the CRAY T3D emulator release. For a discussion of specific issues related to this release, see section 4, page 65.

The emulator version of PVM is much more limited than the other two versions. Specifically, it can only be used to communicate between tasks on the CRAY Y-MP system, and only in a multitasking environment. PVM applications running under the emulator cannot communicate with other PVM executables running elsewhere in the network.

The emulator version uses CRAY Y-MP multitasking and shared memory for communication. This is extremely fast, and it is possible to develop CRAY Y-MP applications to run under the emulator that are not intended to be ported to CRAY T3D systems. However, the primary goal of the emulator version is to support programming on the CRAY T3D system.

Using PVM on Cray Research systems 1.2

You may choose to use PVM to communicate between processes on a number of different computer systems. With PVM available on all types of Cray Research 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 Cray Research systems, this large number of combinations can be grouped into six 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 an executable running on a single PE of a CRAY T3D system. (On a CRAY T3D system, there is one task per PE.)
application	The entity running in a single partition of a CRAY T3D system. An application is made up of a number of tasks all running the same program.
process	The entity running on the UNICOS operating system or another UNIX system.

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

1. One application running on a CRAY T3D system. (PVM uses high-performance connections between the tasks on the PEs in the partition.) This scenario requires only the Cray MPP version in stand-alone mode.

2. Two processes running on a single CRAY Y-MP system. (PVM uses networking capabilities for communication.) This scenario requires the network version, or, in some cases, the emulator version.

3. One process running on a CRAY Y-MP system and another application running on an associated CRAY T3D system. (PVM connects across the channel between the two systems.) This scenario requires both the network version and the Cray MPP version.

4. Two applications running on separate partitions on a single CRAY T3D system. (PVM uses network communications through the associated CRAY Y-MP system.) This scenario requires both the network version and the Cray MPP version.

5. One process running on a CRAY Y-MP system and a second process running on a second CRAY Y-MP system. (PVM is used across the network between the two systems.) This scenario requires the network version.

6. One process running on a CRAY Y-MP system and a second process running on a workstation. (This involves using PVM across the network connecting the systems.) This scenario requires the network version.

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 for use on a Cray Research 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. A strategy appropriate for the fast communications available between CRAY T3D PEs may not, for example, be appropriate between a CRAY Y-MP system and a workstation.

ORNL PVM releases
1.3

The three Cray Research PVM releases are based on PVM releases from the Oak Ridge National Laboratory (ORNL). The network version includes much of the source from an ORNL release with a few Cray Research changes. The Cray MPP version is predominantly Cray Research code with some code from an ORNL release. The emulator version is all Cray Research code. The ORNL release syntax is defined by user-guide and man-page documentation that is included in the ORNL release.

The Cray MPP and network versions of PVM documented in this manual are based on ORNL release 3.3.4. The emulator version, to which no new features are being added, is based on ORNL release 3.2.4. Since ORNL releases do not coincide with Cray Research releases, the two may not synchronize with one another. This scheduling has several ramifications:

- You should not try to use PVM libraries or commands built from the ORNL source with those built from Cray Research releases in the same application.
- Functions or changes introduced by ORNL into new versions may not appear in Cray Research versions until the next major release.
- Cray Research often adds these new ORNL features in a minor release if the features are seen as important and can be added without adversely impacting existing applications. Such changes may not be documented until the next documentation update, usually at a major release.

Finally, Cray Research recognizes the importance of maintaining compatibility between Cray Research and ORNL versions of PVM. This will, on occasion, mean that new releases are not upward compatible with earlier releases if the ORNL release is not upward compatible, or that extensions introduced by Cray Research may be superseded by new ORNL features with a different user interface. Cray Research documents such differences and, if possible, tries to make the transition to the new version easier.

Cray MPP Version [2]

This section describes the operation of PVM on Cray MPP systems. On the CRAY T3D system, which contains up to 2048 processing elements (PEs), some subset of this number are assigned to a job running on the system. Those PEs are collectively known as a *partition* .

The Cray MPP implementation of PVM can be used in either or both of the following modes:

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

There is one PVM library that is part of the compiler environments for the Cray MPP system. When the Cray MPP executable 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.

Features and differences
2.1

This section summarizes special features that can be found in the Cray MPP version of PVM and notes differences between it and the other versions. These features are also documented in the applicable PVM man pages in appendix D, page 155.

PE number
2.1.1

Most existing MPP 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 MPP programming, the Cray MPP version lets you use PE numbers in place of pvm_tids in many of the PVM functions. An extra function, `pvm_get_PE`(3), returns the PE number associated with a pvm_tid.

Global group
2.1.2

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 CRAY T3D system, a static, well-defined group consisting of all the tasks (or PEs) in the MPP 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 nil `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.

Own PE number
2.1.3

CRAY T3D 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)
```

Number of PEs
2.1.4

CRAY T3D 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`.

`PvmDataInPlace` *semantics*
2.1.5

The Cray MPP version of PVM treats data buffers packed using `PvmDataInPlace` encoding differently than the network version does. In the Cray MPP 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.

PVM channels
2.1.6

The Cray MPP version of PVM contains a special communications interface called *PVM channels*. Basically, two PEs establish a channel, in which there is a single source data buffer and a matching destination data buffer. Each transfer request specifies a channel, and the transfer occurs at very high rates. For more information about the channel functions, see subsection 5.12, page 81.

Environment variables
2.2

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 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 subsection describes those that you can set by using UNICOS environment variables. For more information about the `pvm_setopt` function, see appendix D, page 155, or use the `man`(1) command to view the man page online.

When setting an environment variable for the Cray MPP version, you must ensure that it is available for the CRAY T3D executable. If you are using the Cray MPP version in stand-alone mode, this means the environment variable must be set before the executable is run:

```
% setenv PVM_TRACE 7
% ./t3d.out -npes 16
```

If you are using PVM in distributed mode, the PVM daemon starts the CRAY T3D executable. Therefore, you must set the environment variable before the daemon is started:

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

Remember , the CRAY Y-MP daemon starts the CRAY T3D executable, not the task that calls `pvm_spawn`(3).

PVM_AUTO_ERR
2.2.1

The `PVM_AUTO_ERR` environment variable sets the PVM error -handling value, which is equivalent to the `PvmAutoErr` option in `pvm_setopt`.

Setting this option with `PVM_AUTO_ERR` lets you do so without changing your source.

PVM_CHECKING
2.2.2

Certain common PVM operations run the risk of losing data or causing the application to abort. By default, PVM performs checks to avoid these problems. While the costs of these checks are not prohibitive, they can have an impact on performance, and they may be unnecessary for your application. The `PVM_CHECKING` environment variable lets you control whether the checks are performed. This control is at a very gross level: all the checks are performed or none are performed.

The specific checks controlled by `PVM_CHECKING` include the following:

- 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.

- When data is unpacked, a CRAY T3D remote load operation is performed. In certain situations, an operand range error can result. The most common occurs when dynamically allocated heap data is used with `PvmDataInPlace` encoding. By default, PVM checks and makes sure that no such operand range error occurs.

If you set `PVM_CHECKING` to 0, neither of these checks is performed. If you set `PVM_CHECKING` to 1 (the default setting), both of the checks are performed.

PVM_DATA_BUFFERS
2.2.3

The `PVM_DATA_BUFFERS` environment variable sets the initial and incremental number of send buffers. For more information on send buffers, see subsection 2.3, page 15.

PVM_DATA_MAX
2.2.4

The `PVM_DATA_MAX` environment variable sets the integer number of the maximum number of bytes in an initial message. It 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 CRAY T3D 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.

PVM_MAXGTIDS
2.2.5

The PVM_MAXGTIDS environment variable changes the maximum number of tasks that can join a group. For information about the out-of-resources error, PvmOutOfResGmems, see subsection 2.4.3, page 25.

PVM_MAX_PACK
2.2.6

The PVM_MAX_PACK environment variable sets the initial and incremental data block sizes. For information about setting this variable, see subsection 2.3.3.3, page 19, and subsection 2.3.3.4, page 20.

PVM_PE_LIST
2.2.7

The PVM_PE_LIST environment variable lists the PEs that should communicate with the daemon. The PEs are specified in a comma-separated list. For information about setting this variable, see subsection 2.6.1.2, page 28.

PVM_RETRY_COUNT
2.2.8

The PVM_RETRY_COUNT environment variable sets the number of times that PVM retries sending a message to another PE before giving up and returning a PvmOutOfResSMP error. The default is 500. See subsection 2.4.1, page 23, for more information.

PVM_SM_POOL
2.2.9

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. The default value is the larger of the following values:

- Two times the number of PEs
- 10

For information about the out-of-resources error, PvmOutOfResSMP, see subsection 2.4.1, page 23.

PVM_TOTAL_PACK
2.2.10

The PVM_TOTAL_PACK environment variable establishes the upper limit on memory allocated for send buffer data blocks. For information about setting this variable, see subsection 2.3.3.5, page 20.

PVM_TRACE
2.2.11

The PVM_TRACE environment variable sets a mask of trace options, equivalent to the PvmTraceOpts options in pvm_setopt(3). Setting these options using PVM_TRACE 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.

Buffer memory management
2.3

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 subsection discusses these topics.

Basic design
2.3.1

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 the 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.

2.3.2 Simple scenario, part 1

The following scenario shows PVM memory usage, using the default settings.

Table 1. Default settings for buffer memory management

User call	PVM action	Memory use (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
`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

Table 1. Default settings for buffer memory management
(continued)

User call	PVM action	Memory use (sending PE)
`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 memory is allocated by PVM 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.

Controlling memory use 2.3.3

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

All five parameters can be set 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`, 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` to change specific cases.

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

Initial number of send buffers 2.3.3.1

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 paramenter. You can call `pvm_getopt`(3) with the `PvmDataBuffers` option to find out the value of `PVM_DATA_BUFFERS`.

Send buffer increment 2.3.3.2

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:

```
setenv PVM_DATA_BUFFERS 10+4
```

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

Send buffer initial size
2.3.3.3

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

Send buffer increment size
2.3.3.4

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

Total memory use
2.3.3.5

PVM tracks the amount of memory allocated for data blocks, both the 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` to change the setting dynamically. You can also use this option with `pvm_getopt` to see total memory use. To see how much memory is remaining from the current limit, use the `PvmTotalPackLeft` option with `pvm_getopt`.

Simple scenario, part 2
2.3.4

In the original scenario (section 2.3.2, page 16), 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` 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` once with `PvmMaxPack` and perhaps again with `PvmMaxPackIncr`, each set to the preceding values, prior to the preceding sequence.

Out-of-resources errors
2.4

When running a PVM application on Cray MPP systems, you may receive out-of-resources errors. Receiving one of these errors, shown in Table 2, means that you have encountered a fixed limit within the PVM implementation.

Table 2. 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. 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 may 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-resources 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 subsections briefly discuss each limit, describe how to raise it, and identify ways to use PVM more efficiently.

PvmOutOfResSMP
2.4.1

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 and give 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 subsection 2.2.8, page 14. Many applications will find 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 subsection 2.2.9, page 14.

If you are hitting this 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 this 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.

PvmOutOfResBuf
2.4.2

The PvmOutOfResBuf error occurs only if you have set the send buffer increment parameter to 0 (see subsection 2.3.3.2, page 18, 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, default 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 it 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 runs faster due to the reduced number of function calls.

PvmOutOfResGmems 2.4.3

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` with the `PvmMaxgtids` option (if this is done, the function must be called on each PE before any groups are formed).

Remember that the Cray MPP 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 Cray MPP system. This will simplify your code, and you will get better performance when performing barriers across the group or broadcasts to the group.

Data types and the Cray MPP version 2.5

This subsection describes how PVM data types are implemented in the Cray MPP version. This discussion assumes that you are familiar with the functions used to pack and unpack data; see subsection 5.8, page 76, and subsection 5.9, page 78, for more information.

Table 3 presents basic information about data types available in the Cray MPP version.

Table 3. Data names: Cray MPP version

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`

16-bit Fortran data types
2.5.1

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 CRAY T3D system does not support these data types.

32-bit Fortran data types 2.5.2

On the CRAY T3D 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.

64-bit integer data 2.5.3

In the network version of PVM, data conversion of integers is limited to 32 bits of accuracy. The Cray MPP 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 subsection 2.2.2, page 12.)

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

Distributed mode 2.6

The following subsections discuss several issues specific to the distributed mode of the Cray MPP version. Using this mode requires that you use both the Cray MPP and the network versions. If you are not familiar with the network version, you may want to read section 3, page 41, before reading this subsection.

The following discussion assumes that the application you are running is using only the CRAY T3D system and the coupled CRAY C90 or CRAY Y-MP system. This assumption is made for simplicity only; your application can use other CRAY T3D partitions, other Cray Research systems, or other systems connected to your network. Most of the same issues still apply.

Major issues 2.6.1

The following subsections discuss several key issues related to the distributed mode.

Architecture and host system 2.6.1.1

The CRAY T3D system uses the PVM daemon that runs on the coupled CRAY Y-MP system. Consequently, when you create a host file or specify a system in a `pvm_spawn` call, you can treat the CRAY T3D system and the CRAY Y-MP system as the same. If your application needs only a CRAY T3D system and a CRAY Y-MP system, your host file contains a single entry: the system name of the CRAY Y-MP system.

PE communication 2.6.1.2

PEs on the CRAY T3D system communicate with the daemon and other PVM tasks by using sockets. 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 CRAY T3D 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 cost of communications between PEs. Because much of socket communication is single-threaded in the agent process, the PVM daemon, or UNICOS itself, the cost goes up as more PEs try to communicate at the same time.

For these reasons, by default only PE0 establishes communications with the daemon, and Cray Research suggests you 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` and the application to read, and both must read the same value.

Note

At present, PE0 always establishes communications with the daemon, even if PE0 is not specified in `PVM_PE_LIST`. It is suggested that `PVM_PE_LIST` specify PE0 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.

Cray MPP executables
2.6.1.3

To spawn an executable on the CRAY T3D system, you should ensure (perhaps by specifying `PvmTaskHost` or `PvmTaskArch` when calling `pvm_spawn`) that it will be started on a Cray Research system that is attached to a CRAY T3D system. The daemon on that system looks at the executable and determines if it is a CRAY T3D or a CRAY Y-MP executable.

When you build your CRAY T3D executable, you can optionally fix the number of PEs at load time. For such executables, the `pvm_spawn` *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, you have a *plastic* executable. For these, the `pvm_spawn` *count* parameter specifies the number of PEs that you want for the executable.

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

CRAY T3D tasks
2.6.1.4

The CRAY T3D task must be spawned by the PVM console or by another PVM task. You cannot simply start the daemon and then start the CRAY T3D task. The CRAY T3D task looks for special information passed to it by the daemon, and, if the task fails to find this information, it assumes it is in stand-alone mode.

This limitation may be lifted in future releases.

Cross-system dynamic groups
2.6.1.5

You cannot form a dynamic group consisting of tasks from the CRAY T3D and CRAY Y-MP systems or another system. You must view group handling on each system as being completely independent. If the CRAY T3D 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 CRAY T3D task to `MYGROUP` sends messages only within the Cray MPP system; no messages will go outside that system.

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

Session example 2.6.2

The following subsections describe the programs and commands that you can use to run a distributed application involving the CRAY T3D system. You can do this a number of different ways; the following is an example of one way.

Parent task 2.6.2.1

The parent task runs on the CRAY Y-MP system and uses PVM as described in section 3, page 41. The key line of interest is the call to `pvm_spawn`. 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 (`mpp.a.out`) is built with a fixed PE count (for example, using the `mppldr -X` option), then `nproc` must be larger than or equal to the PE count, and `count` returns the PE count. If the executable is built as a plastic executable (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 Research system, the code assumes that any Cray Research 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.

CRAY T3D task 2.6.2.2

The task on the CRAY T3D 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, and you cannot use the channel send and receive mechanisms. You must also be careful that only those PEs that can communicate with the daemon try to do so.

Both of these constraints can be dealt with 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.

Command script
2.6.2.3

This section takes you through a series of shell and PVM console commands that could be used in a typical, short PVM session. First, set the environment variables that are needed by the PVM executable, the daemon, or both. You can then can start the daemon. This sequence assumes that the network PVM executables are located in a directory that is part of your `PATH` environment variable.

The following commands set several environment variables and then start the PVM daemon, using a host file that was previously set up, and runs it as a background UNIX process. Because the tasks started by the daemon inherit the environment of the daemon, these variables must be set now.

```
% setenv PVM_AUTO_ERR 2
% setenv PVM_PE_LIST 0,4
% pvmd3 hostfile &
```

While you can simply start the parent executable at this step, having the console available enables you to monitor progress of the application. The following command starts the PVM console:

```
% pvm
```

As noted by the `pvm>` prompts, the following commands are console commands.

The following console command starts the parent task:

```
pvm> spawn parent
```

Because this scenario assumes that only one system is specified in the host file (the CRAY Y-MP coupled to the CRAY T3D system), it is not necessary to specify system or architecture information in this step.

The following console command shows the tasks known to the daemon:

```
pvm> ps
```

Initially, you see only the parent task listed, but once the `pvm_spawn` is executed, you should see tasks appear for the CRAY T3D system. You can recognize these tasks by the following, which look different from those of the parent:

- Name of the executable
- pvm_tids

If your application completes, all the task entries should disappear. If one task finishes or aborts, its entry should disappear. However, if your application deadlocks or hangs, you cannot distinguish it from the normal executing case by using the console `ps` command. In that case, you must use other UNICOS or UNICOS MAX commands.

The following command terminates your PVM session:

```
pvm> halt
```

When this command is issued, all active tasks are killed, along with the daemon and your PVM console. This is the recommended way to stop the daemon; other techniques (such as using a UNICOS `kill` command) may leave temporary files that you will have to clean up manually.

Performance considerations 2.6.3

The following subsections describe how to efficiently use PVM for communication between a CRAY T3D system and the associated CRAY C90 or CRAY Y-MP system The ability to use PVM to distribute applications between a CRAY T3D system and CRAY C90 or CRAY Y-MP system is a powerful one. It provides the opportunity for greater overall performance for applications, as well as the ability to incrementally port applications to the CRAY T3D system. However, you must be aware of the performance obtainable for such data transfers and design your applications accordingly.

Remember that performance is affected by the number of PEs that are trying to communicate with the CRAY C90 or CRAY Y-MP system. Performance is also affected by other users on the CRAY C90 or CRAY Y-MP system. If your system is heavily loaded, your performance will be slower and may vary significantly.

This subsection begins by describing the basic use of PVM for communication between the two systems. It then describes different strategies that can help improve performance, including the following:

- Establishing a direct task-to-task connection
- Increasing PVM fragment sizes
- Combining I/O and PVM
- Data conversion

This subsection provides rough estimates for performance, in terms of both latencies and bandwidths. These estimates should be used as approximations, as differences in system and software configurations, system load, and other factors have major effects on the performance that you obtain.

Performance overview 2.6.3.1

The three key approaches to using PVM for distributed communications provide different levels of performance. Table 4 summarizes these and provides estimates of their relative data transfer performance, for large data transfers.

Table 4. PVM distribution methods

Technique	Bandwidths	Description
Basic PVM	0 to 3 Mbyte/s	See subsection 2.6.3.1.2, page 35.
Direct PVM	0 to 10 Mbyte/s	See subsection 2.6.3.1.3, page 35 and subsection 2.6.3.1.4, page 36.
Combined PVM and I/O	5 to 100 Mbyte/s	See subsection 2.6.3.1.5.

The wide range of bandwidths reflects the variability likely due to differences in system load, system configuration, and other parameters, many of which are outside the control of an ordinary user .

Latencies for the three techniques are comparable, from 5 to 10 milliseconds per message. Subsection 2.6.3.1.1 discusses the reasons for this relatively high latency.

None of these performance results take data conversion into account. Subsection 2.6.3.1.6, page 38, discusses issues related to data conversion.

System calls and PVM 2.6.3.1.1

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.

These 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 see if data can be output.

The I/O-related system calls from the CRAY T3D system are very slow. These system calls are carried out by the agent on the CRAY Y-MP system. For PVM, the basic cost of these system calls accounts for the bulk of the time spent in sending messages to and from the CRAY T3D system. Issues such as channel speed and data conversion are relatively insignificant, compared to system call overhead. The key to obtaining good performance is to focus on reducing the number of system calls.

In addition, the cost of system calls goes up as more PEs try to make them at the same time. By default in distributed mode, only PE0 communicates with the PVM daemon, but additional PEs can also be permitted to communicate (for more information, see subsection 2.6.1.2, page 28). If you are interested in performance, think very carefully before using more than one PE to make PVM calls outside the CRAY T3D system. This guideline applies regardless of the other options discussed in this subsection.

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

Basic PVM 2.6.3.1.2

The simplest way to use PVM in distributed mode is to use it as it is used in stand-alone mode. You write code that creates send buffers, packs the data, and sends it, and you ensure that the *pvm_tid* specified in the `pvm_send` call is that of a task outside the CRAY T3D system. This approach works correctly, but results in very poor performance. Bandwidths of a few megabytes per second should be expected.

This communication strategy results in poor performance for several reasons:

- By default, when a message is sent, data is packed into 32,768-byte fragments by PVM and sent to the PVM daemon in 32,768-byte fragments. (This 32,768-byte default is a UNICOS parameter.)
- Two CRAY T3D system calls are required to send each such fragment of data from the CRAY T3D system. The daemon forwards this data to the CRAY Y-MP task. Most of the time is taken up by the two CRAY T3D system calls.
- Receipt of a message follows a similar path, except that three system calls are required to read each fragment.

If only a few, short messages are going to be transferred, in parts of the application that are not time critical, this basic PVM works well. However, if more or longer messages are being transferred, optimizations should be considered.

Direct PVM 2.6.3.1.3

To obtain better performance, first consider whether direct task-to-task routing is advisable. With this option, two tasks using PVM establish a direct socket connection, bypassing the PVM daemon. Alone, this option does not improve performance significantly but, when used with other options, it can.

The drawback to this strategy is that a socket represents an open file descriptor. If more than one PE is being used to communicate outside the CRAY T3D system, each PE must be able to open another file descriptor for each such connection. This may rapidly deplete the number of available open files.

This option is enabled by the `pvm_setopt` option `PvmRoute`. By default, all PVM tasks passively accept a direct routing request from another task, but they will not actively request one. To actively request these, make the following call:

C:

```
pvm_setopt (PvmRoute, PvmRouteDirect);
```

Fortran:

```
CALL PVMFSETOPT (PVMROUTE, PVMROUTEDIRECT,
OLDVALUE)
```

As long as this option is active, task-to-task transfers from this task request a direct socket connection with the other task. While you need only make this call from one end of the communication, it is probably cleaner to make the call from both the CRAY Y-MP and the CRAY T3D tasks. You should make this call before doing any communications.

PVM fragment sizes
2.6.3.1.4

The second key step to obtaining good performance is to increase the PVM fragment size. Most of this fragment is used for user data, but a small portion is used for internal PVM headers. You can increase this fragment size so that more data is sent or received per system call. This change noticeably increases performance.

The drawback to this strategy is increased memory. When these fragment sizes are increased, PVM dynamically allocates more CRAY T3D memory when sending and receiving messages. If your application is tight on memory, this can be important. More memory is allocated in the agent on the CRAY Y-MP system, meaning that it also uses more memory when running.

You can increase fragment size with the `pvm_setopt` option `PvmFragSize`. You specify the byte size that you want for each fragment. The calls are as follows:

C:

```
pvm_setopt (PvmFragSize, 100000);
```

Fortran:

```
CALL PVMFSETOPT (PVMFRAGSIZE, 100000, OLDVALUE)
```

Unlike `PvmRouteDirect`, `PvmFragSize` has only a local effect; so you should call it from both the CRAY T3D and the CRAY Y-MP ends of the application. Also, when the direct task-to-task socket is opened, it uses this value in setting the internal socket buffer size, which is very important. Therefore, you should make this call before doing any communications.

By combining direct routing and increased fragment sizes, you should obtain a bandwidth speedup of two to three times for large transfers of a megabyte or more.

I/O and PVM 2.6.3.1.5

By using PVM alone, your performance is limited. PVM uses I/O, but does so relatively inefficiently. This inefficiency is not due to poor design, but is inherent in doing interprocess communication with sockets. Other mechanisms are possible, but are less flexible or more complex.

Using the CRAY T3D system, however, users have obtained very high bandwidths for doing simple I/O requests, where a single system call can quickly drive data across the high-speed channel. Using multiple CRAY T3D I/O gateways, several such transfers can occur at once.

A relatively simple way to achieve good performance while using PVM is to separate the two parts of a PVM message: control information and data. You can use PVM to send the control information and regular UNICOS MAX I/O features to send the data. To do this, write the data to a file and use PVM to indicate to the other task that the data is ready.

This type of update can be easy to do. Consider the following basic PVM send sequence:

```
pvm_initsend (PvmDataDefault);
pvm_pkbyte (bigbuf, tests[j], 1);
pvm_send (ptid, 668);
```

You could change this to the following, which uses I/O to transfer the data:

```
pvm_initsend (PvmDataDefault);
f2 = open (filename, O_RDWR);
write (f2, bigbuf, tests[j]);
close (f2);
pvm_send (ptid, 668);
```

In this example, `filename` represents an existing file, the name of which is known to both the sender and receiver. A total of three system calls transfer the data, regardless of the amount of data.

The receiving task does something very similar:

```
pvm_recv (-1, 668);
f2 = open (filename, O_RDWR);
read (f2, bigbuf, tests[j]);
close (f2);
```

These simple code sequences, using a disk file in `/tmp`, can give a performance boost of 5 to 10 times over basic PVM. In addition, the code still follows a standard message-passing paradigm and is therefore portable to other systems with shared file systems.

There is additional room for improvement. The file need not be a disk file; you could use the SSD or memory-resident files. Continually reopening and reclosing the file may not be necessary . And there are other options that you can set when the file is opened that can adjust buffer sizes and control the manner in which I/O is performed.

The drawbacks here are twofold. Another file descriptor is being used and file space must be allocated. The other has to do with data conversion.

Data conversion
2.6.3.1.6

When you specify `PvmDataDefault` when calling `pvm_initsend`, PVM converts the data between IEEE and Cray Research formats for all forms of typed data. This is not done very efficiently on the CRAY Y-MP end.

Data conversion can be done efficiently on the CRAY Y-MP system, however, by using the data conversion functions available in the UNICOS Fortran libraries (see the *UNICOS Fortran Library Reference Manua*l, publication SR–2079). 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, then call `CRAY2IEG` or `IEG2CRAY` as appropriate on the data you are about to write from the CRAY Y-MP system or have just read from the CRAY T3D system.

If you are using file I/O, another, 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 *I/O User's Guide* , publication SG–3075, and the *Advanced I/O User's Guide* , publication SG–3076.

Network Version [3]

This section describes the networking implementation of the Parallel Virtual Machine (PVM).

Virtual machine
3.1

In the network version, 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 process initiation, message transmission and reception, and synchronization by using 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.

Applications and environments
3.2

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, while the support tools and interactive graphic facilities are valuable as an easy-to-use and convenient concurrent programming environment. PVM, along with HeNCE, has been used for molecular dynamics simulations, superconductivity studies, distributed fractal computations, matrix algorithms, and as the basis for teaching concurrent programming.

PVM program development
3.3

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

1. Add PVM function calls to your application for process initiation, communications, and synchronizing. For syntax descriptions of these functions, see section 5, page 71.

2. Build executables for the systems that you will use. For information on building PVM executables, see subsection 3.3.2, page 43.

3. Define the virtual machine by creating a host file as described in subsection 3.3.3, page 46.

4. Execute the PVM daemon and your application as described in subsection 3.3.4, page 51 and subsection 3.3.7, page 58.

5. Troubleshoot the application, if necessary. For information on PVM troubleshooting, see subsection 3.3.8, page 58.

6. Optimize the application as described in subsection 3.3.9, page 60.

Using process identifiers
3.3.1

All processes that enroll in PVM are represented by an integer task identifier, or pvm_tid. Because pvm_tids 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)
```

Building PVM executables
3.3.2

Once you have added PVM function calls, build executables for each architecture and system on which they run. Usually, all that you need to add are the references to the PVM libraries. The references for Cray Research systems and for systems of other companies are shown in Table 5.

Table 5. PVM libraries

System	Library	Contents
Cray Research	`libpvm3.a`	All PVM functions
Other	`libpvm3.a`	Basic PVM functions
	`libfpvm3.a`	Fortran interface to PVM functions
	`libgpvm3.a`	PVM group functions

On Cray Research systems, a simple PVM code could be linked as follows:

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

On other systems, a link may be more complex:

```
cc -o compute -l/tmp/pvm/libpvm3.a -l/tmp/pvm/libfpvm3.a compute.o
```

Contact your system administrator for information on where the PVM libraries are installed, and consult documentation for the appropriate system for information on commands and options used in compiling programs.

After this is done, install the executables on the systems you will be using. If you set the `ep` option in the host file for a system, install it in that directory. Otherwise, install it in the following:

`$(HOME)/pvm3/bin/`*arch*

The appropriate PVM name is specified by *arch*, as shown in Table 6.

This table lists PVM names for different systems, along with other information you may need. PVM may also run on architectures not listed here. Contact your system administrator to determine whether PVM has been built for the system you want to use.

Table 6. PVM system names

PVM name	System (architecture)	Notes
`AFX8`	ALLIANT FX/80	
`ALPHA`	DEC Alpha	DEC OSF/1
`ALPHAMP`	DEC Alpha multiprocessor	DEC OSF/1 3.0 or later ††
`BAL`	Sequent Balance	DYNIX
`BFLY`	BBN Butterfly TC2000	
`BSD386`	80386/486 UNIX box	BSDI
`CM2`	Thinking Machines CM-2	Sun front-end system
`CM5`	Thinking Machines CM-5	†
`CNVX`	CONVEX C-series	
`CNVXN`	CONVEX C-series	Native mode
`CRAY`	CRAY T3D, CRAY C90, CRAY Y-MP, and CRAY X-MP systems	
`CRAY2`	CRAY-2 systems	

† MPP system with native port available
†† SMP (scalable multiprocessor) system with shared memory-port available

Table 6. PVM system names (continued)

PVM name	System (architecture)	Notes
CRAYSMP	CRAY S-MP systems	
CSPP	CONVEX Exampler SPP	
DGAV	Data General Aviion	
HP300	HP 9000, model 300 (Snake)	HP-UX
HPPA	HP 9000, PA–RISC	
I860	Intel iPSC/860	You add `link -lrpc` to loader libraries when using `libpvm.a.`†
IPSC2	Intel iPSC/2 386 host	
KSR1	Kendall Square	††
LINUX	80386/486	Linux
MASPAR	Maspar/Dec MIPS front end	
MIPS	MIPS	
NEXT	NeXT	
PGON	Intel Paragon	`link -lrpc`†
PMAX	DEC/MIPS architectures (3100, 5100, and so on)	
POWER4	IBM Power-4	††
RS6K	IBM RISC System/6000	

† MPP system with native port available
†† SMP system with shared memory-port available

Table 6. PVM system names
(continued)

PVM name	System (architecture)	Notes
`RT`	IBM RT	
`SGI`	Silicon Graphics IRIS	You add `-lsun` to loader libraries when using `libpvm.a`. If using YP, you may have to add `-lsun` to `ARCHDLIBS` in the makefile.
`SGI5`	Silicon Graphics IRIS	OS release 5.0 or later
`SGIMP`	Silicon Graphics IRIS multiprocessor	†
`SUN3`	Sun-3	Works fine under SunOS 4.*x*.
`SUN4`	Sun-4, SPARCstation, and so on	Supports SunOS 4.*x*.
`SUN4SOL2`	Sun-4, SPARCstation, and so on	Supports Solaris (SunOS 5.*x*).
`SUNMP`	Sun-4, multiprocessor	Includes Cray CS6400†
`SYMM`	Sequent Symmetry	
`TITN`	Stardent Titan	
`UVAX`	DEC/MicroV AX	
`UXPM`	Fujitsu	UXP/M
`VCM2`	Thinking Machines	CM-2/DECV AX front end

Formatting host files 3.3.3

All systems in the PVM virtual machine must have entries in the host file, one per line. Lines beginning with a hash symbol (#) (possibly preceded by white space) are ignored.

† SMP system with shared memory-port available

If you do not want PVM to start a host immediately, but you may start it later 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 7, you should include it in the host file, preceded by the ampersand (`&`) character.

Figure 1 contains the names of sample systems, the basic information needed in a host file.

```
# my first host file
thud
fred
wilma
barney
betty
```

Figure 1. Simple host file

The master `pvmd3` for a group is started by hand on the local system, and it starts slaves on each of the remaining systems.

You should verify that the system on which the master `pvmd3` will run (the *master host*) is included in the file and that no system is listed more than once.

You can specify the options listed in Table 7 after any system name in the host file.

Table 7. 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 is ~/$PVM_ROOT/lib/debugger.
dx=*loc*	Specifies a location for pvmd3 other than the default, ~/$PVM_ROOT/lib/pvmd3. Use this option for Cray Research systems or for other systems on which PVM has been installed in a central location. This option is useful in debugging new versions of PVM. *loc* may be a simple file name, an absolute path name, or a path relative to the user's home directory on the remote system. pvmd3 is installed in /usr/bin/pvmd3 when the network version is installed on Cray Research systems, so you will likely need to set dx=/usr/bin/pvmd3 in a host file for Cray Research systems.
ep=*paths*	Specifies a series of paths to search for the requested files to spawn on the system. 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 $HOME/pvm3/bin/$PVM_ARCH: $PVM_ROOT/bin/$PVM_ARCH, where PVM_ARCH is the PVM name of an architecture in Table 6, page 44.
lo=*userid*	Specifies an alternate login name for the system. The default is the login name on the master system.

Table 7. Host file options
(continued)

Option	Description
`so=ms`	Causes the master `pvmd3` to request that you manually start a `pvmd3` on a slave system when the `rsh`(1) and `rexec`(1) network services are disabled but IP connectivity exists. The default is false. (This option is rarely used.) You cannot start the master system from the PVM console or background when you are using this option.
`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 follows: `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 are using this option.
`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 1000000. The default is 1000. (This option currently has no effect on PVM operation.)
`wd=`*path*	Specifies a working directory in which all spawned tasks on the host will execute.

A dollar sign ($) in an option introduces a variable name, for example *$PVM_ARCH* . Names are expanded from environment variables by each PVM daemon.

The simple host file in Figure 1, page 47, works well if you have a login with the same name on the other four systems and the local system in your `.rhosts` file on each of them. To supply an alternative login name for a system (`NAME`, in the following example), add the following to its host file entry:

```
thud lo=NAME
```

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

```
cyclone so=pw
```

You can also specify the path of the daemon executable on any system, as follows:

```
gust.cray.com dx=/usr/bin/pvmd3
```

The string given is passed to a shell so that variable expansion works, in the path, as shown in the following example:

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

Note

By default, `pvmd3` is installed in `/usr/bin/pvmd3` on Cray Research systems; therefore, you must specify the `dx` option when a system is a Cray Research system.

Each of the preceding options has a default value (for example, the current login name for `lo` or `/tmp/pvm/pvmd3` for `dx`). You can change any of these for a given host file by specifying them on a line with an asterisk (`*`) in the host field, as follows:

```
* lo=afriend so=pw
```

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

You can override the location of executable files by adding the following to your host file entries:

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

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

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

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

Figure 2. Host file with host name options

Starting and stopping the daemon
3.3.4

Once you have written a host file, you can start up the master `pvmd3` by passing it the host file as an argument. You must specify the appropriate path for `pvmd3`. On a Cray Research system, for example, enter the following:

```
/usr/bin/pvmd3 hostfile &
```

On other computer systems, the `pvmd3` path is likely to be the following:

```
/tmp/pvm/pvmd3 hostfile &
```

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

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

To shut off PVM, enter `halt` at a PVM console prompt. For detailed information, see subsection 3.3.6.

If the master `pvmd3` has trouble starting a slave `pvmd3` on a system, the error message from the master `pvmd3` may indicate the problem.

Managing application output 3.3.5

All output written to standard output or standard error by any of your PVM executables is not written to your terminal, but is instead sent to the master PVM daemon. The daemon, in turn, prefaces each line with a pvm_tid denoting its source, and writes it to a file named `/tmp/pvmd.`*uid* . *uid* is your UNIX user ID for the system on which you started the daemon. 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 file, you can request that output be sent as a PVM message to another task. For more information, see the `PvmOutputTid` and `PvmOutputCode` options of the `pvm_setopt`(3) function.

You can also redirect output by using options on the console `spawn` command (see Table 8, page 54) or with the `pvm_catchout`(3) function.

Using the PVM console 3.3.6

The PVM console is another PVM executable that lets you monitor and modify the virtual machine. The `pvm` command starts the console, which can be started and stopped multiple times on any of the systems on which PVM is running.

Starting the console
3.3.6.1

The PVM console is started by using the following command line:

/usr/bin/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:

```
% /usr/bin/pvm
pvmd already running
pvm>
```

If the daemon is not started, the pvm command tries to start one, but the command must first find the daemon. (Currently, the pvm 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 is the following.

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 with 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 Cray Research systems:

  ```
  % mkdir ~/pvm3
  % mkdir ~/pvm3/lib
  % ln -s /usr/bin/pvmd3 ~/pvm3/lib/pvmd
  ```

On Cray Research systems, if pvmd3 is not found or cannot be executed, the pvm command explicitly tries to start /usr/bin/pvmd3. This special processing is not performed for other systems.

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

  ```
  % /usr/bin/pvm
  pvm>
  ```

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

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

Using console commands
3.3.6.2

When you enter the pvm(1) command, the console responds with a prompt and accepts the commands described in Table 8. While running PVM, you can list the other available commands by executing the `help` command.

Table 8. 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 the 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 commands. If you enter `help` followed by a command name, a brief description of the syntax is displayed.
`id`	Prints the pvm_tid of the console. (The console is simply another PVM task.)

Table 8. Console commands
(continued)

Command	Description
`jobs [-l]`	Displays a list of running jobs. The `-l` option provides more detailed output.
`kill [-c]` *tasks*	Kills a PVM user process. The `-c` option indicates that children of the tid(s) 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) This example illustrates `nhost` usage: `ps -ngust` This command requests the status of a system named `gust.`
	`-l` Shows long output
	`-x` Shows console task

Table 8. Console commands (continued)

Command	Description
	`ps` output includes the following fields: `HOST` System executing 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. May be one or more of the following: `a` Task is waiting 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 a single 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 `pvmds` are left in an idle state.
`setenv [`*name* `[`*value*`]]`	Displays or sets environment variables.
`sig` *num task*	Sends signal to tasks.

Table 8. Console commands
(continued)

Command	Description
`spawn` [*options*] *file* [*args*]	Starts a PVM application.
	`-`(*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*
	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`	Sets or displays trace event mask. The *names* argument refers to names defined in the PVM include file:
`trace [+]` *names*	`/usr/include/pvm3/pvmtev.h`
`trace [-]` *names*	
`trace [+] *`	
`trace [-] *`	
`unalias` *name*	Undefines command alias.
`version`	Displays `libpvm` version.

Running PVM applications 3.3.7

Once `pvmd3` is running successfully, you can execute your application. Run the application on one of the systems in the PVM virtual machine by using the following command:

```
./a.out
```

You can also use the PVM console `spawn` command, as follows:

```
pvm> spawn a.out
```

Troubleshooting PVM 3.3.8

This subsection addresses common problems encountered when using PVM and their solutions. There are several kinds of problems that can keep `pvmd3` 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 type (to the shell), enter the following command:

`rsh hostname` *daemon*

Replace *daemon* with the location of the PVM daemon (such as `/tmp/pvm/pvmd3` or `/usr/bin/pvmd3`). On some systems, including UNICOS, 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.

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

PVM already running 3.3.8.1

When you start `pvmd3`, you may receive a message that PVM is already running because a file exists in `/tmp`. If no `pvmd3` is already running, it is likely that the last time you used PVM you did not terminate `pvmd3` with the console `halt` command. Remove the file named in the message and start `pvmd3` again.

Permission denied
3.3.8.2

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 different from that you have specified. To determine the name of your local system on the remote system, execute `telnet`(1) or `rlogin`(1) 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 you connected to) thinks is the name of your local system (the one on which you entered `telnet`). Make sure you put that system name (the full name, not just the first 16 characters) in your `.rhosts` file on the remote system. 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.

Login incorrect
3.3.8.3

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.

Version incorrect
3.3.8.4

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.

Other problems
3.3.8.5

If you get any other messages, make sure your `.cshrc` file on the remote system is not either printing something out when you log in or trying to set your terminal characteristics (usually with 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:

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

Optimizing use of PVM 3.3.9

Several PVM functions are particularly useful when developing applications that involve Cray Research systems. This subsection 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 subsections will improve performance but will make the application harder to move to different PVM virtual machines or other systems.

Using Cray Research multiprocessor systems 3.3.9.1

When multiple PVM processes run on a Cray Research multiprocessor , the processes use sockets 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 executables, this is probably your only choice.

In some applications, the PVM processes each execute the same executable, 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.

You should seriously consider using Autotasking whenever your application is run on a Cray Research system. The communications overhead for Autotasking is orders of magnitude less than that for sockets, even on the same system, so you may be better off having only one fully Autotasked PVM process on the Cray Research system. In many cases, you may 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 Cray Research system.

Using PVM direct routing 3.3.9.2

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 user process 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` function.

Avoiding XDR conversion 3.3.9.3

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:

- `PvmDataRaw`
- Transfer bytes
- User-controlled conversion

`PvmDataRaw` 3.3.9.3.1

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 if they do not.

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

Transfer bytes 3.3.9.3.2

When data is transferred as bytes, no data conversion takes place. The `pvm_pkbyte`(3) and `pvm_upkbyte`(3) functions bypass data conversion because byte data is defined to be typeless. 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, 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).

User -controlled conversion 3.3.9.3.3

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 Cray Research formats and IEEE, IBM, DEC, and other formats.

For example, one process (perhaps a workstation) could use `pvm_pkbyte` to pack the data; the receiving process (the Cray Research system) could use `pvm_upkbyte` 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 taking place between a Cray Research system and another system for which data conversion functions are available. (For descriptions of conversion functions, see the *UNICOS Fortran Library Reference* , publication SR–2079.)

Data types in the network version 3.4

This discussion of how PVM data types are implemented in the network version assumes that you are familiar with the functions used to pack and unpack data. For more information about these functions, see subsection 5.8, page 76, and subsection 5.9, page 78.

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

Table 9 presents basic information about data types on Cray Research systems, including the CRAY Y-MP and CRAY-2 systems, but not including the CRAY T3D systems or the Cray Research SUPERSERVER products.

Table 9. Data names: network version

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`
(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`

Fortran data types
3.4.1

Table 9 does not contain some Fortran 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.

64-bit integer usage
3.4.2

The network version of PVM does not support the `INTEGER8` specification. You must specify `INTEGER4` for Cray Research 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.

Network environment variables
3.5

In the network version, you may use the following environment variables to customize your PVM environment.

The PVM_ROOT environment variable specifies the path where PVM libraries and system programs are installed. This variable is set on each PVM system in order for PVM to function. There is no default value.

The PVM_EXPORT environment variable names the environment variables that a parent task exports to its children through the pvm_spawn(3) function. Multiple names must be separated by a colon. There is no default value.

The PVM_DEBUGGER environment variable specifies the debugger script to use when pvm_spawn(3) is called with PvmTaskDebug set. The default is $PVM_ROOT/lib/debugger.

The PVM_DPATH environment variable specifies the path of the pvmd3(1) command or the start-up script. The default is $PVM_ROOT/lib/pvmd. You can override this setting by using the dx=*loc* option in the host file.

If you use a shell (such as .cshrc) that does not automatically execute a start-up script that sets PVM_ROOT on added hosts, you can set PVM_DPATH to the absolute or relative path of the pvmd start-up script, such as /usr/local/pvm3/lib/pvmd or pvm3/lib/pvmd, the script automatically sets PVM_ROOT.

Emulator Version [4]

This section describes the user-level interface for the Parallel Virtual Machine (PVM) message-passing library for the CRAY T3D emulator. PVM provides a small, efficient set of functions, usable from either the Fortran or C programming languages, supporting the passing of messages between tasks.

PVM operations
4.1

The intent of the emulator is to allow CRAY T3D application developers to use a CRAY C90 or CRAY Y-MP system to develop codes for the CRAY T3D system. The emulator supports code written in Fortran or C. Fortran codes may be written in all styles defined in the Fortran programming model.

PVM applications for the emulator should be written as SPMD (same program, multiple data) applications. That is, one program should be written to execute on each processing element (PE) of the CRAY T3D system, and it should use PVM to send data to and receive data from other PEs. In many cases, these programs will operate in an even simpler, SIMD (single instruction, multiple data) style of programming, in which each PE executes exactly the same code. This code can be written in Fortran or C.

The emulator runs message-passing (PVM) applications with any even number of PEs, less than or equal to 32.

In practice, the limiting factor on the types of codes that can be emulated is likely to be CRAY Y-MP memory size or UNICOS user database (UDB) limits on memory usage and the number of processes.

The emulator library collects and outputs performance statistics. For PVM, it provides information on the number of messages and the amount of data transferred (sent and received) for each pair of PEs that communicates using PVM. In addition, estimated CRAY T3D timings for these transfers are output (although this is currently based on a very primitive model). This data is output in a form that can be processed by the

UNICOS `procview`(1) command. As with other emulator instrumentation, output of this data is optional. At the same time, the memory reference statistics can be used to verify that each PVM task is referencing only data local to each PE.

Before using the software, make sure your system has the following include files. These files are located in `/usr/include/emulib`.

- For C programs, the `pvm3.h` file provides prototypes of the PVM functions and definitions of the constants.
- For Fortran programs, the `fpvm.h` file provides definitions of the constants (as Fortran parameters).

For more information on the emulator, see the *CRAY T3D Emulator User's Guide*, publication SG–2500.

Data types in the emulator version
4.2

This discussion of how PVM data types are implemented in the emulator version assumes you are familiar with the functions used to pack and unpack data. For more information on these functions, see subsection 5.8, page 76, and subsection 5.9, page 78.

Data type support in the emulator version is slightly different from the support in the other two versions. If you are careless, this can lead to problems later, especially when you move your code from the emulator to the CRAY T3D system.

Table 10 presents basic information about data types for the emulator version.

Table 10. Data names: emulator version

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

Table 10. Data names: emulator version
(continued)

Data characteristics	C functions	Fortran names
64 bits, floating-point	`pvm_*pkdouble`	`REAL8`
(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`

Other data types
4.2.1

Missing from Table 10 are the C functions and Fortran names that map to 32-bit data on the CRAY T3D system. These omissions are intentional. The CRAY Y-MP system does not support 32-bit data, and if you use these data types in the emulator , you will actually operate on 64-bit data items.

If you choose to use these data types, you must do so very carefully; it is very easy to create a program that runs successfully on the emulator but produces incorrect results when moved to the CRAY T3D system.

64-bit integers and the emulator
4.2.2

In the emulator version, no data precision is lost with 64-bit integers packed with `PvmDataDefault`. This is different from the network and Cray MPP versions and can lead to different results.

Emulator differences
4.3

The following subsections summarize special features that can be found in the emulator version of PVM and note differences between it and the other versions. These features are also documented in the applicable PVM man pages in appendix D, page 155.

PE numbers
4.3.1

Most existing MPP applications and algorithms are written to use PE numbers for communication. Standard PVM notation supports only the concept of a pvm_tid, whose internal representation is subject to change. To simplify CRAY T3D programming, the emulator version allows PE numbers to be used in place of pvm_tids in many of the PVM functions. An extra function, `pvm_get_PE`(3), returns the PE number associated with a pvm_tid.

Global group definition
4.3.2

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 the CRAY T3D system, there is a static, well-defined group consisting of all the tasks (or PEs) in the MPP partition. This group 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 nil `char` pointer) to refer to this global group. This allows the PVM implementation to quickly realize that it can use some key optimizations to carry out barriers and broadcasts.

pvm_get_PE *usage*
4.3.3

CRAY T3D 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)
```

Getting the number of PEs
4.3.4

CRAY T3D 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 file `fpvm3.h`.

PVM channels 4.3.5

The CRAY T3D channel functions are supported in the emulator version, but there is no performance benefit.

Unsupported PVM functions 4.3.6

The following PVM functions are not supported in the CRAY T3D emulator implementation. In the emulator, the PVM machine is of a fixed size and composition (a single, emulated CRAY T3D partition), and all PEs are running the same program. As a result, the following PVM functions do not apply and are not supported:

- `pvm_addhosts`(3)
- `pvm_config`(3)
- `pvm_delhosts`(3)
- `pvm_disptrace`(3) (Cray Research extension; CRAY T3D systems only)
- `pvm_kill`(3)
- `pvm_mstat`(3)
- `pvm_notify`(3)
- `pvm_perror`(3)
- `pvm_sendsig`(3)
- `pvm_setopt`(3)
- `pvm_spawn`(3)
- `pvm_tasks`(3)
- `pvm_tidtohost`(3)

Calling these functions is permitted, but, if called, each returns an error status (`PvmNotImpl`).

Because no new features are being added to the emulator version, other PVM functions listed in this manual are also not supported in the emulator version. Calling these functions results in unsatisfied external references at load time.

Obsolete functions 4.3.7

Since the initial release of the emulator version of PVM was made available, several functions have been renamed or removed. These functions are generally still supported in the emulator , but a warning message is output whenever an application uses the old name, indicating that it must be changed. These old interfaces will not be available on the CRAY T3D system; so you should heed the messages and change your code as soon as possible.

Functions and Subroutines [5]

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

- Basic operations
- Task control
- Option management
- Trace management
- Dynamic system configuration
- Dynamic task group management
- Data transmittal
- Data receipt
- Barrier synchronization
- Global operations
- Channel management
- Signaling
- Error handling

General information
5.1

This section 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 normally use them to perform the tasks.

In most cases, each logical PVM function is represented by a C and C++ function and a Fortran subroutine. For more information about a specific man page, see appendix D, page 155, or use the `man`(1) command to view the man page online. To simplify references, this discussion refers to C functions, C++ functions, and Fortran subroutines as functions unless individual differences require documentation.

PVM-detected errors 5.1.1

For a complete list of the PVM error messages and the value associated with each, see appendix A, page 135. 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 `pvm_setopt`(3) for a description of the `PvmAutoErr` option.

PVM include files 5.1.2

Each of the three Cray Research PVM releases contains the same two include files, `pvm3.h` and `fpvm3.h`. In theory, the include files are the same. In practice, however, due to differences in when each is released by Cray Research and when (or if) each is installed at your site, the include files show minor differences.

To avoid confusion, the include files are installed in different directories, as follows:

`/usr/include/mpp`	Cray MPP version
`/usr/include/pvm3`	Network version
`/usr/include/libemu`	Emulator version

While Cray Research will make every effort to ensure that constant values are the same across the three files, you should make sure you are using the correct version. You can do this by referring to the proper subdirectory in the `#include` compiler control line, as follows:

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

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

```
#include <pvm3.h>

cc -I /usr/include/mpp
```

Note

PVM include files may exist in `/usr/include` if your site has installed the public domain version of PVM, or they may be there from earlier releases of the CRAY T3D emulator. Be careful not to use those files by mistake.

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; so a null character in the middle of the character sequence, which is valid in Fortran, terminates the string.

Basic operations 5.2

You can perform basic operations with the functions in Table 11.

Table 11. Basic operations functions

C and C++ function	Fortran subroutine	Description
`pvm_get_PE`	`PVMFGETPE`	Converts a *pvm_tid* into a PE number (Cray Research 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
`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

Task control
5.3

You can control PVM process creation and termination with the task control functions in Table 12.

Table 12. 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

Option management
5.4

You can control PVM options with the functions in Table 13.

Table 13. Option management functions

C and C++ function	Fortran subroutine	Description
pvm_setopt pvm_getopt	PVMFSETOPT PVMFGETOPT	Sets a PVM option or returns its current value

Trace management
5.5

You can control PVM traces with the function in Table 14.

Table 14. Trace management function

C and C++ function	Fortran subroutine	Description
`pvm_disptrace`	`PVMFDISPTRACE`	Displays PVM trace buffer (Cray Research extension)

Dynamic system configuration
5.6

The dynamic system configuration functions, described in Table 15, 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 15. 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 on tasks

Dynamic task group management
5.7

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 assigned by the user. All tasks wanting to join a group must specify the same character string.

Dynamically joining and leaving a group must be done with care. Race conditions 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.

You can use the dynamic task group functions in Table 16 to do these tasks.

Table 16. Dynamic task group management functions

C and C++ function	Fortran subroutine	Description
`pvm_joingroup` `pvm_lvgroup`	`PVMFJOINGROUP` `PVMFLVGROUP`	Joins or leaves a dynamic group
`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

Data transmittal
5.8

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 17 to do these tasks.

Table 17. Data transmittal functions

C and C++ function	Fortran subroutine	Description
pvm_psend	PVMFPSEND	Packs and sends data in one call.
pvm_initsend	PVMFINITSEND	Initializes a send buffer.
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_bcast	PVMFBCAST	Broadcasts a message to all tasks in a group.
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_setsbuf	PVMFSETSBUF	Specifies a new buffer as the current send buffer.
pvm_getsbuf	PVMFGETSBUF	Returns the buffer identifier of the current send buffer.

Data receipt 5.9

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:

- 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.

Once 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.)

You can use the data receipt functions in Table 18 to perform these tasks.

Table 18. Data receipt functions

C and C++ function	Fortran subroutine	Description
`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_trecv`	`PVMFTRECV`	Receives a message with a time-out.
`pvm_bufinfo`	`PVMFBUFINFO`	Returns information about a message.

`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).

`pvm_setrbuf`	`PVMFSETRBUF`	Specifies a new buffer as the current receive buffer.
`pvm_getrbuf`	`PVMFGETRBUF`	Returns the buffer identifier of the current receive buffer.
`pvm_freebuf`	`PVMFFREEBUF`	Releases receive buffers. See `pvm_mkbuf`(3).
`pvm_recvf`	(Not applicable)	Supplies a user-written comparison function.

Barrier synchronization
5.10

The barrier function in Table 19 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 using the same barrier.

Table 19. Barrier synchronization function

C and C++ function	Fortran subroutine	Description
`pvm_barrier`	`PVMFBARRIER`	Creates a barrier to synchronize tasks

Global operations
5.11

The functions in Table 20 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 20. Global operation 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

Channel management (Cray Research extension) 5.12

A special feature has been added to the PVM interface for the CRAY T3D system. This feature, called *channels* , can be used by many applications to obtain the maximum possible performance for data transfers on the CRAY T3D system. It combines the high performance available on the CRAY T3D system for transferring data by using shared memory, while maintaining the private memory model programming style to which message-passing programmers are accustomed.

This feature is most useful for applications that depend upon regular , iterative communication. For example, in each iteration, each PE sends the same amount of data from the same starting address to the same PE, which receives it into the same starting address.

With regular PVM message-passing syntax, this information is provided to the PVM library each time a message is transferred. This limits the possible performance, because the system cannot know in advance that regular behavior is occurring. By using the channels feature, an application can provide this information to PVM one time and initiate fast transfers repeatedly thereafter . The process is as follows:

- Both the sending and receiving PEs call special functions to set up a channel between them; this process is slow but only needs to be performed once.

- The application can then call functions to initiate a transfer, using a syntax similar to that for the send and receive functions. It also is necessary that the two PEs be synchronized. This can be done using channel features, or, if synchronization is already being done in the algorithm, the transfer can make use of it implicitly.

You can use the channel functions in Table 21 to perform these tasks.

Table 21. Channel functions

C and C++ function	Fortran subroutine	Description

`pvm_channel_src_setup`
`pvm_channel_dst_setup`
`pvm_channel_sync`
`pvm_channel_send`
`pvm_channel_send_ok`
`pvm_channel_recv`
`pvm_channel_close`
	`PVMFCHSRCSETUP`
	`PVMFCHDSTSETUP`
	`PVMFCHSYSNC`
	`PVMFCHSEND`
	`PVMFCHSENDOK`
	`PVMFCHRECV`
	`PVMFCHCLOSE`
		Sends and receives messages over a channel. See `pvm_channels`(3).

Signaling
5.13

The functions in Table 22 supports signals of different kinds with PVM tasks.

Table 22. 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

Error handling
5.14

The function in Table 23 provides simple help for handling PVM-generated errors.

Table 23. Error handling function

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

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

Part 2: HeNCE

Part 2 describes the Heterogeneous Network Computing Environment (HeNCE). The contents of this section are based on the contents of a report entitled *HeNCE: A User's Guide, Version 2.0* by Adam Beguelin, Jack Dongarra, G. A. Geist, Robert Manchek, Keith Moore, Peter Norton, and Vaidy Sunderam.

Part 2 is organized as follows:

Section	Title and description
6	*HeNCE Overview* contains general product information.
7	*Setting Up HeNCE* describes the installation of the product and the customization of your environment.
8	*Using HeNCE* describes the steps involved in making HeNCE a useful tool.
9	*Tutorial Example* describes how to build a simple, parallel dot-product program.

HeNCE Overview [6]

HeNCE is a graphical parallel programming environment based on the X Window System environment. HeNCE provides an easy-to-use interface for creating, compiling, executing, and debugging parallel programs that run under the network version of PVM (described in section 3, page 41). HeNCE programs can be run on a single UNIX workstation or over a network of heterogeneous systems.

HeNCE greatly simplifies writing parallel programs. Using HeNCE, you explicitly specify parallelism between subroutines by drawing a graph, as shown in Figure 3, where nodes in the graph are subroutines written in either Fortran or C.

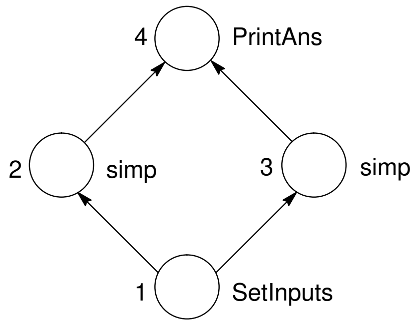

Figure 3. Sample HeNCE program

You must supply the HeNCE graph and the source code for the subroutine nodes. HeNCE then automatically executes the subroutines in parallel (whenever possible) across a network of heterogeneous systems. You do not need to use any parallel programming primitives or write any code beyond that supplied for the node subroutines.

HeNCE relies on the PVM system for process initialization and communication. If you want to write explicit message-passing parallel programs on a network of systems, you should explore using the PVM system directly. HeNCE is designed for programmers and scientists who want to exploit a collection of systems to which they have access, yet do not want to delve into the details of parallel programming. HeNCE allows programmers unfamiliar with parallelism to combine sequential subroutines with a picture, creating a parallel program.

Feedback on the product
6.1

The goal of the HeNCE project is to develop parallel programming tools that assist scientists, researchers, and students in utilizing parallel computers. The project designers encourage people to try HeNCE and to give them feedback. If you have suggestions or problems regarding HeNCE, you may send email to `hence@cs.utk.edu`. Suggestions and problems may also be reported to Cray Research, Inc.

Definitions
6.2

This subsection defines the terminology used in the HeNCE environment.

Term	Definition
HeNCE	The entire programming environment and paradigm.
`htool`	The graphical interface that realizes the HeNCE programming environment.
graph	The HeNCE graph consists of nodes representing subroutines and arcs representing data dependencies. Individual nodes are executed under PVM. Special node constructs are used to control the flow of execution in the graph.
node program	A HeNCE procedure (implemented in HeNCE node language) to describe the computation to be performed and the node subroutine (implemented in C or Fortran) used to perform the computation.
node subroutine	A procedure, written in either C or Fortran, which is invoked by a node program in a HeNCE graph. The procedures can be taken from existing code. A node subroutine can be invoked from multiple node programs.

A complete HeNCE program consists of the HeNCE graph, node programs, and node subroutines.

Setting Up HeNCE [7]

To create a HeNCE program, you must run `htool` and its associated programs. The executable files `htool`, `proxyd`, and `master` must be in your path on the system on which you write your HeNCE program. These executables need not be on each system on which your HeNCE program will execute.

Note

Contact your local system administrator to learn the names of the directories in which the HeNCE software is installed.

Using the default environment
7.1

While you can use HeNCE in other ways, it is simplest if you adopt the following default HeNCE environment:

- Your working environment consists of a network of computers running UNIX.
- You can enter the `rlogin`(1) or `rsh`(1) command from one computer on your network to another without entering a password. If this is not the case, you may want to enter the command `man rhosts` and read the UNIX man page about the `.rhosts` file. Also, see the section on nonstandard PVM environments below.
- You place all HeNCE and PVM software within default directories. HeNCE stores executable images for your parallel programs within directory `$(HOME)/pvm3/bin/$(PVM_ARCH)`. `PVM_ARCH` is the environment variable that defines the name of the system architecture on which you are running. For information about PVM system names, see subsection 3.3.2, page 43.

Customizing HeNCE
7.2

There are two mechanisms by which you may customize `htool` operation and appearance:

- Attributes related to the X Window System, such as colors and fonts, are defined as X resources.
- Other attributes are set in a `.htool` file.

Most users will have no need to customize `htool` using either mechanism since all important values have reasonable defaults on most common architectures.

Using a `.htool` *file*
7.2.1

Many HeNCE attributes are set by adding lines to a file called `.htool`, which is placed in your home directory. Each line contains a attribute name and a value in the following form:

NAME: *value*

Table 24 lists all attribute names that can be used in your `.htool` file. All of the attribute names are uppercase and must be followed by a colon.

For example, the following is a small, but complete `.htool` file. It sets both the language for node programs and the editor that you use to edit programs.

```
EDITOR_COMMAND: xterm -title %s -e mg %s
```

Table 24. `.htool` file attributes

Attribute	Description
`EDITOR_COMMAND`	The command `htool` uses when you want to edit a node program or node subroutine. Emacs users can set it to `emacs %s` since emacs can create its own window when running under the X Window System. For additional information, see subsection 7.2.1.1, page 90. The default is the following: `xterm -title %s -e vi %s`
`LANGUAGE`	The language used for your node subroutines. The value is either C or Fortran. The default is C.
`PRINT_COMMAND`	The command HeNCE uses to print a Postscript image of your graph. The default is the following: `lpr %s`
`TMP_DIR`	The UNIX directory where `htool` scribbles temporary cost matrices and so on. The default is the following: `/tmp`
`PVM_HOST_FILE`	Name of PVM host file when `pvm3d` is started. There is no default. For a list of PVM system names that you can use, see Table 6, page 44. For additional information, see subsection 7.2.1.2, page 90.
`SUB_DEFS_FILE`	Name of file from which `htool` reads function prototypes for node subroutines. `htool` automatically determines this information from the actual parameters passed in node programs. `SUB_DEFS_FILE` allows you to override `htool` type determination. It is unlikely that you will need to use this feature. There is no default.
ARCH `_CC`	A printf-like string that says how to compile a C node program. *ARCH* is a PVM architecture name such as `HPPA`, `RS6K`, or `SUN4`. The default depends on the architecture. For additional information, see subsection 7.2.1.3, page 91. If not otherwise defined, the following is assumed. `cc -o %o %{-g}%[-O] %D %I %s %w %L -lm`

Table 24. .htool file attributes
(continued)

Attribute	Description
ARCH _FC	A printf-like string that says how to compile a Fortran node program. Escapes are the same as for *ARCH* _CC. ARCH is an architecture name. There are defaults for HPPA, SUN4, PMAX, and RS6K. The default depends on the architecture. For additional information, see subsection 7.2.1.3, page 91. If not otherwise defined, the following is assumed. `cc %{-g}%[-O] %D %I -c %w %%f77 -o %o %{-g}%[-O] fmain.f %W %s %L -lm`
*ARCH_* BUILD_FLAGS	*ARCH* is an architecture name. BUILD_FLAGS is a string that contains any characters described in Table 25. There are defaults for HPPA, RS6K, PMAX, and SUN4. No flags are automatically included for other architectures. The default depends on the architecture. For additional information, see subsection 7.2.1.4, page 92.

EDITOR_COMMAND 7.2.1.1

When composing a HeNCE program, htool starts various editing sessions to allow you to edit node subroutines and node programs. The editor that htool uses for these editing sessions is set by the EDITOR_COMMAND attribute.

PVM_HOST_FILE 7.2.1.2

Before executing a HeNCE program, htool starts a virtual system. This is done by creating a PVM host file and starting a PVM daemon. The PVM host file is created using defaults from a master PVM hosts file. For instance, you may have a different login name on your Cray Research system than you do on your workstation. This can be set in the master PVM host file so that, whenever htool starts a virtual machine (including the Cray Research system), it will use the correct login name. The location of this master PVM host file is set by using the PVM_HOST_FILE attribute, and the default is /dev/null. If you leave this resource unspecified, htool does not return an error but simply builds a generic PVM host file. If specified, PVM_HOST_FILE names a file containing a list of PVM-style host entries. These entries are used by htool to tell the local pvmd3 how to start up pvmd3 processes on remote systems. The format of this file is as follows:

hostname [*options*]

hostname The name of the remote system as used in `htool`.

options Options can be used to change the following:

- The method for starting the remote `pvmd3` is started (whether by the `rsh`(1) command or, if the `so=pw` option is specified, by the `rexec`(3) library routine).
- The user name to use on the remote system (if different from the user name on the local system).
- The location of the remote `pvmd3` program. If no `PVM_HOST_FILE` is specified, or if a system specified in the `htool` cost matrix does not appear in the file specified by `PVM_HOST_FILE`, `htool` tells the local `pvmd3` to use its default settings when starting up remote `pvmd3` processes. For a compete description of the PVM host file options, see Table 7, page 48.

ARCH _`CC` and *ARCH*_`FC` 7.2.1.3

ARCH _`CC` is a printf-like string that says how to compile a C node program; *ARCH* _`FC` is a printf-like string that says how to compile a Fortran node program. In both cases, *ARCH* is a PVM architecture name such as `HPPA`, `RS6K`, or `SUN4`. Special escapes include the following:

```
%{something}  include "something" only if debugging
%[something]  include "something" only if NOT debugging
%D            represents predefined "-D'' preprocessor defines
%I            represents predfined ''-I'' includes
%L            represents predefined ''-L'' and ''-1'' libraries
%o            name of object file (foo)
%s            name of source file (foo.c or foo.f)
%w            name of wrapper (cw_foo.c or fw_foo.f)
%W            name of compiled wrapper (cw_foo.o or fw_foo.o)
```

`htool` has special wired in values for `HPPA`, `PMAX`, `RS6K`, and `SUN4`, which can be overridden here.

If you wish to create a custom *ARCH* _CC, you do it by adding to the basic string. You should continue to use the escapes that represent the items that are known to be required. For example, to add a new library to the system you are using (system name CRAY), you add a new -l to the basic string:

```
CRAY_CC: -o %o %{-g}%[-O] %D %I %s %w %L -lm -lsomenewlib
```

*ARCH_* BUILD_FLAGS
7.2.1.4

ARCH is an architecture name. The BUILD_FLAGS is a string which contains any character described in Table 25.

Table 25. *ARCH_* BUILD_FLAGS string

Character	Description
_	Appends an underscore to function name to call from Fortran
U	Specifies uppercase function names to call from Fortran
l	Adds extra arguments at the end of argument lists to give the length of Fortran CHARACTER arrays

Customizing X resources
7.2.2

Because htool is based on the X Window System, you can customize its behavior by setting X resources. X resources are set by placing a line in your .Xresources file and entering the following command:

```
xrdb -load .Xresources
```

You may want to add the xrdb(1) command to your .xinitrc file so that it runs automatically when you start the X Window System.

For example, if you put the following lines in `.Xresources`, they would cause most things to be drawn in blue.

```
htool.notReadyColor: blue
htool.readyColor: blue
htool.doneColor: blue
htool.runningColor: blue
```

The resources `brokenColor`, `deadColor`, `doneColor`, `errorColor`, `idleColor`, `normalColor`, `notReadyColor`, `readyColor`, `runningColor`, and `warningColor` are used to specify colors of the nodes when drawing graphs in compose and trace mode.

The degree to which `htool` can be customized may seem overwhelming at first. However, all X resources have reasonable default values and if your installation of PVM and HeNCE is standard, no changes to these resources should be necessary.

The X resources and their defaults are listed in Table 26.

Table 26. `htool` X resources

Resource	Example value	Description
`titleBackground`	`cyan`	Specifies background color for title bars.
`titleForeground`	`black`	Specifies foreground color for title bars.
`brokenColor`	`red`	Indicates one the following: `htool` cannot configure the system, the system died, or the `proxyd` program died.
`deadColor`	`black`	Indicates that the node program finished and exited.
`doneColor`	`magenta`	Indicates the build finished compiling or the node completed execution.
`errorColor`	`red`	Indicates an error in a node program.
`idleColor`	`yellow`	Indicates that a compilation is in progress or that a system, node, or subroutine is waiting for requests from children.
`normalColor`	`white`	Specifies default background color for cost matrix.

Table 26. htool X resources
(continued)

Resource	Example value	Description
notReadyColor	grey	Indicates that node is not ready to execute.
readyColor	yellow	Indicates that node is ready to execute.
runningColor	green	Indicates that a build is in progress or that a system, node, or subroutine is currently running.
warningColor	orange	Indicates that there is a warning about the HeNCE graph for a particular node.

Using HeNCE [8]

Once HeNCE is installed and configured, you are ready to create a parallel program with it. This section describes the process.

Getting started 8.1

To create, compile, and run a HeNCE program, you perform the following steps:

1. Execute `htool` and the main `htool` window appears on your screen. See subsection 8.2, page 97.
2. Select the working directory you will use. See Table 28.
3. Draw a picture (graph) that shows the desired parallel structure. See subsection 8.3, page 99.
4. Annotate the nodes in the graph. Annotations are entered into a pop-up editor window. Figure 4 shows the annotation for node 2 of Figure 3.

```
NODE [ 118 257 ] 2
 < double a;
 < double mid;
 < int n;
  s1 = simp(a, mid, n);
 > double s1;
```

Figure 4. Annotation for node 2

The annotation shows the call to the sequential subroutine as well as what variables from predecessor nodes are read and what variables are made available to successor nodes. `a`, `mid`, and `n` variables are set by node 1, and variable *s1* is read by node 4. See subsections 8.4, page 100, and subsection 8.5, page 103.

5. Write all of the sequential subroutines that are called from the HeNCE nodes. See subsections 8.4, page 100, and subsection 8.5, page 103.

6. Define a cost matrix. That is, the virtual parallel machine on which the parallel program runs. This is done by specifying a cost matrix that lists the cost of running each of the program' s subroutines on each of the systems you select for use by the parallel computation. The cost matrix does the following:

   - Defines the list of systems on the network that make up the virtual parallel machine.

   - Provides HeNCE with execution cost estimates for each subroutine that help it to run each subroutine on a system that is chosen to maximize overall performance. The default cost matrix runs all nodes on the system on which you are running `htool`. See subsection 8.6, page 117.

7. Select `Build` from the Build menu to compile the parallel program. See subsection 8.7, page 119.

8. Select `Run Program` from the Run menu to execute the parallel program.

   HeNCE animates the program's execution and produces a strip chart that shows the utilization of the system in the virtual parallel machine during the run. See subsection 8.7, page 119.

9. Save your graph and cost matrix files.

After your program has executed, you may want to exit `htool`. See subsection 8.8, page 122. If you may want to trace the program run and print the HeNCE graph, see subsection 8.9, page 122, and subsection 8.10, page 123.

Executing `htool` 8.2

The `htool` command initiates the Heterogeneous Network Computing Environment (HeNCE), a software environment based on the X Windows System and designed to assist in developing parallel programs that run over a network of heterogeneous computers using PVM facilities. HeNCE is based on a parallel programming paradigm where an application program can be described by a graph. HeNCE is composed of integrated graphical tools for creating, composing, executing, and analyzing HeNCE programs.

The `htool` interface consists of a single large window, shown in Figure 5.

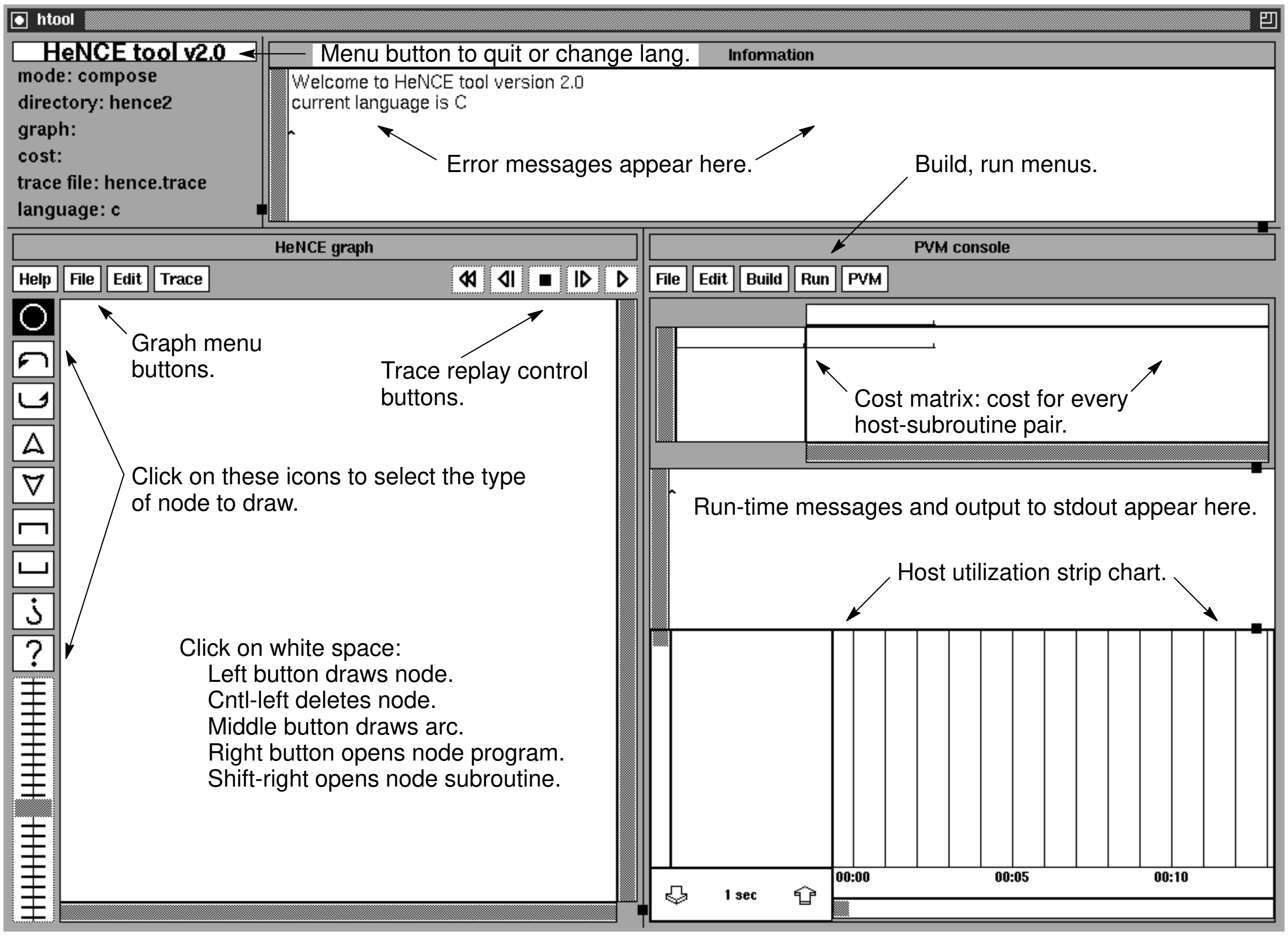

Figure 5. `htool` window

This window is divided into the segments described in Table 27:

Table 27. The `htool` interface

Segment	Description
Top left	Displays status information such as the current UNICOS directory, the current graph, cost matrix, and trace file names, and the currently selected language. The `HeNCE tool v2.0` button is at the top of this segment.
Top right	Displays error messages, warnings, and so on. This segment is entitled Information.
Lower left	Draws and displays the HeNCE graph. This segment is labeled HeNCE graph. The graph menu buttons, trace replay control buttons, and icons are located along its borders.
Lower right	Displays the following information about the PVM virtual machine: cost matrix and strip chart showing HeNCE host utilizations. This segment is labeled PVM console. The build, run menus are located on the top border .

To quit `htool`, click the left mouse button on the `HeNCE tool v2.0` button in the top left segment. A menu with three options appears. Select `Quit` from the options described in Table 28:

Table 28. `HeNCE tool v2.0` menu

Menu option	Description
`Change Directory`	Changes the directory in which `htool` looks for and saves files.
`Change Language`	Changes the language (C or Fortran) used to express the sequential subroutines. All nodes in a HeNCE program must call subroutines written in the same language.
`Quit`	Exits the HeNCE environment. Before you quit, save any changes you have made to graphs or cost matrices or they will be lost.

Drawing graphs 8.3

HeNCE graphs are drawn in the large initially empty segment in the lower left of the `htool` window. You use the mouse to draw nodes and arcs. The default mouse actions for drawing in compose mode are shown in Table 29.

Table 29. Default mouse actions in compose mode

Mouse button	No shift or control	Shift	Control
Left	Add node	Move node	Delete node
Middle	Add arc	Move arc	Delete arc
Right	End node program	End node subroutine	

To add a node to the graph, select the type of node you want to add and click the left mouse button in the compose window. To delete a node, hold the CONTROL key and click the left mouse button on or near the node to delete. To move the node, hold the SHIFT key down and use the left mouse button to click and drag the node to its new location.

To draw arcs between nodes, click the middle button on the node at the tail of the arc, drag the mouse to the node at the head of the arc, and release the middle button. To delete arcs, hold the CONTROL key and click the middle button on or near the arc. An arc may be moved by holding the SHIFT key while clicking the middle button near one end of the arc, dragging it to another node, and releasing the middle button.

To edit the node program for a particular node, click the right button near a node. An edit window appears, containing the node program. The node program is parsed immediately after the edit is complete. If there are problems with the node program, error messages appear in the message center window, and `htool` asks you if you would like to edit the node program again. For a description of the syntax for node programs, see subsection 8.5, page 103. The node program is considered a part of the graph and is stored along with the graph.

Note

Changes to the node program are not saved until the graph is stored.

To edit the node subroutine (the actual C or Fortran source code) for a node, hold the SHIFT key while clicking the right mouse button near the node. HeNCE then starts an edit session for that source file. The file name is the name of the function, appended with a `.c` (for C source code) or `.f` (for Fortran source code), depending on the current setting of the language selection (see subsection 7.2.1, page 88).

Using HeNCE nodes
8.4

The HeNCE language contains a number of different node types:

- Computation nodes
- Special-purpose nodes

Each type is represented by an icon shown in Figure 5. All but one of the icons come in pairs. This subsection provides an overview of the types available; subsection 8.5, page 103 describes how you use these types in HeNCE node programs.

Computation nodes
8.4.1

The circle icon represents a sequential computation node. It calls a user-supplied subroutine when it executes. This node type represents the basic unit of parallelism in HeNCE. Sequential computation nodes may execute when all of their predecessors in the graph have executed. The icons that come in pairs represent nodes.

A *node program* is the annotation you enter for a node. For computation nodes, it consists of a description of what variables the node uses and what subroutine it calls. A computation node's node subroutine is the text of the subroutine it calls.

If you click the right mouse button on a computation or special-purpose node, an editor window that displays it's node program appears. If you shift-click the right mouse button on a computation node, an editor window that contains the node subroutine appears.

Simply exit the editor normally (`ZZ` for vi) to make these windows go away. If you wish to use an editor other than the default editor, `vi`, see subsection on customizing HeNCE.

Special-purpose HeNCE nodes 8.4.2

HeNCE provides constructs that denote four different types of control flow:

- Loop
- Fan
- Pipe
- Conditional

These constructs allow you to specify dynamic control flow, which depends on parameters at run time. The constructs are defined by special-purpose nodes. There is a begin node and an end node for each construct. For instance, there are begin-loop and end-loop nodes for defining a subgraph making up a loop body. The special-purpose nodes all come in pairs, and must contain a HeNCE subgraph. In simple terms, constructs can be nested but not interlaced. The constructs and special purpose nodes are shown in Table 30.

The four constructs can be thought of as graph rewriting primitives. These constructs add subgraphs to the current program graph based upon expressions that are evaluated at run time.

Table 30. Special purpose HeNCE nodes

Construct	Description
Loop	The loop construct is similar to the conditional in that it specifies a subgraph to be conditionally executed. However , the loop construct also allows iteration on a subgraph as a loop body. In other words, the subgraph making up the loop body is repeatedly added to the program graph based upon a Boolean expression that is evaluated each time through the loop.  BEGINLOOP  ENDLOOP
Fan	The fan construct in HeNCE allows you to specify a parallel fanning out and in of control flow to a dynamically created number of subgraphs. The integer expression attached to the begin-fan node is evaluated to determine how many subgraphs will be created. Each subgraph created by the fan construct executes in parallel.  FANOUT  FANIN
Pipe	The pipe construct in HeNCE provides for pipelined execution of a subgraph. The expression attached to the begin-pipe node indicates whether another data item is to be piped though the subgraph. If the expression evaluates to true, another subgraph is added to the graph in order to execute the additional data item in a pipelined fashion.

Table 30. Special purpose HeNCE nodes(continued)

Construct	Description
	 BEGINPIPE  ENDPIPE
Conditional	Using the conditional construct you may specify a subgraph to be executed conditionally . If the Boolean expression attached to the begin-conditional node evaluates to true, the subgraph contained between the begin-conditional and end-conditional nodes is added to the program graph. If the expression evaluates to false, the contained subgraph is not added.  BEGINSWITCH  ENDSWITCH

Thinking of these constructs as graph rewriting primitives not only provides a mechanism for specifying parallelism but also a natural way of viewing the dynamic parallelism as a graph unfolds at run time.

Annotating nodes
8.5

You must define the precise action of a HeNCE node. For example, a computation node executes a sequential subroutine, but you must specify which subroutine. You do this by creating a textual description called the node program.

Computation nodes 8.5.1

The syntax of node programs varies among HeNCE node types. Computation nodes are the most complex and can serve as the starting point.

Node programs consist of three parts, but not all nodes need all three. You enter these parts in nodes in the following order:

- Input declarations specify which variables the node will use. These variables are defined and set by predecessor nodes in the graph.
- Subroutine call is a call to a sequential subroutine that will be executed when the node executes. The call includes both the subroutine name and all parameters.
- Output declarations define new variables that are declared and set within the node and that are made available to successor nodes in the graph.

Input declarations 8.5.1.1

Input declarations take the following form:

< [*type*] *param-name* [= *expression*] ;

Input declarations state that a node is to find and load the variable *param-name* from its ancestors. If *type* is not specified, the parameter is assumed to exist in ancestor nodes, and it will retain the type declared there. If *type* is specified and differs from the type declared by an ancestor, HeNCE flags an error. If no ancestors contain the parameter, it is created and initialized to *expression* . If there is a type, but no ancestors contain that variable and no *expression* is given, the parameter is created and initialized to a null value.

If a node wants parameters to be loaded from ancestors and to be seen by descendants, it can declare them to be input/output. This is done by substituting <> for < in the preceding syntax. The following examples mean the same as examples 1 through 4, except that, after their subroutine completes, their values can be sent to descendant nodes.

The examples in Table 31 shows the difference between input and input/out input declarations.

Table 31. Input declaration examples

Input	Input/output	Description
`< a;`	`<> a;`	`a` is loaded from ancestors. The type of `a` is inherited from the ancestor's declarations. If `a` does not exist, HeNCE flags an error.
`< int b = 5;`	`<> int b = 5;`	`b` is loaded from ancestors; if `b` does not exist in an ancestor , it is initialized to 5.
`< char c[25] = "foo";`	`<> char c[25] = "foo";`	This example is similar to the previous one, except that `c` is an array of 25 `chars`. If no ancestor has `c`, 25 `chars` are allocated, and the first four are initialized to `foo`.
`< int d[10][15];`	`<> int d[10][15];`	The two-dimensional integer array `d` is retrieved from ancestors. If no ancestor has `d`, the array is allocated, and all of its elements are set to 0.

This is a bit counter-intuitive, because writing the following does not mean that `b` will be initialized to 5 (in other words, if the node's parent sets `b` to 10, then `b` will be 10 and not 5):

```
< int b = 5;
```

Thus, there is an alternate input declaration form:

`NEW '<'` *type param-name* `[ '='` *expression* `] ';'`

This declaration specifies that the parameter will be created anew. Therefore, if a node specifies the following, `b` will be initialized to 5 in this node, regardless of its value (or type) in ancestor nodes:

```
NEW < int b = 5;
```

Input parameters are for input only. They are not sent to descendants. Therefore, when a node declares a parameter for input, it is free to modify that parameter, and those modifications will not be seen by any descendents.

Expressions and default parameters 8.5.1.1.1

It was previously stated that parameters may be initialized to *expressions*. Expressions may also be used to define array dimensions and subroutine arguments. An expression is defined as a combination of constants, parameters, and C operators. For example:

```
5
a
b[3][5] + c * 5.0
(d != 0) ? &d : &e
```

If an expression uses a parameter that is not declared, that parameter defaults to an input/output parameter. Therefore, in the following declaration the first line can be omitted:

```
<> a;
< int b = a;
```

Because `b` mentions `a` in its expression, `a` defaults to an input/output parameter. Even if an expression is not used (for example, `b` is loaded from an ancestor and is never initialized to `a`), the undeclared parameters that it uses still default to input/output parameters.

Circular references are not allowed. The following example is flagged as an error:

```
NEW < int a = b;
NEW < int b = a;
```

Arrays
8.5.1.1.2

Arrays are declared as in C. Each dimension's specification is enclosed in square brackets. These specifications may be expressions involving other parameters. When an array is first declared, it must be declared with a type. This defines the size of the array, which must remain constant throughout the lifetime of the computation. If a node declares an array that an ancestor has already declared, it is an error if the declarations differ in type or in the size of any dimension.

After an array has been defined, nodes may use rectangular portions of it. Thus, nodes can split up an array to perform parallel computations. In the simplest case, a node can specify a single element of an array. For example, if node `1` wants to use element `(5,6)` of integer array `p`, it can declare the following:

```
<> p[5][6];
```

Or it can simply use the following:

```
foo(p[5][6]);
```

Larger , rectangular portions of arrays can also be specified by giving the dimension's low and high values. An empty dimension defaults to the entire contents of the dimension.

The examples described in Table 32 show how you specify portions of a two-dimensional array `A[20][20]`.

Table 32. Two-dimensional array

Portion	Description
`A`	Specifies all of `A`.
`A[][]`	Specifies all of `A`.
`A[0 : 5][10 : 15]`	Specifies the rectangular portion of `A`, starting with `A[0][10]` and ending with `A[5][15]`. This may be used in an expression such as the following: `NEW <> int B[6][6] = A[0 : 5][10 : 15];` In this case, `B` is created, and its values are copied from that rectangular portion of `A`. `B[0][0]` will be equal to `A[0][10]`, and `B[5][5]` will be equal to `A[5][15]`.
`A[0 : 5][11]`	Specifies a 6-element, one-dimensional array whose first value is `A[0][11]` and whose last value is `A[5][11]`.
`A[11][]`	Is a 20-element, one-dimensional array whose first value is `A[11][0]` and whose last value is `A[11][19]`.
`A[11 : 11][0 : 5]`	Is a 1-by-6 two-dimensional array whose first value is `A[11][0]` and whose last value is `A[11][5]`.

You can use these array specifications in expressions, and you can legally specify different portions of the same array. For example, the following declarations are perfectly legal:

```
NEW <> int b[6][6] = A[0 : 5][10 : 15];
NEW <> int c[6][6] = A[1 : 6][10 : 15];
```

Each portion of `A` is considered to be a different I/O parameter. This could be the source of nondeterminacy problems if the partitions of `A` overlap.

The current implementation of HeNCE has been written so that some expressions involving arrays are illegal. In particular, scalar parameters cannot be set to array values. The reason for this has to do with implementation details, and future versions of HeNCE should remove these restrictions.

The following are examples of **illegal** expressions:

```
NEW <> int b = c[5];
NEW <> d[1 : c[5]];
FANOUT 1 i = c[5] TO 10;
foo(d[c[5]]);
```

However , the following are **legal** expressions:

```
NEW <> int b[5] = c[5 : 9];
foo(c[5] + c[6]);
```

More on array passing 8.5.1.1.3

The internal representation of arrays has two ways of storing them:

- As a large, one-dimensional chunk of bytes (which is what you would get in C if you declared an array with constant sizes. For example, `int a[4][5];` gives you a pointer to a chunk of 20 `int`s).
- As a tree of pointers (which is what you get in C otherwise: for example, `char **argv`). When HeNCE allocates an array, it allocates the chunk of bytes, and then builds the pointer-tree on top of it, so you can use either representation.

Currently , HeNCE stores arrays in row major. For Fortran use, HeNCE will need some modification.

A by-product of HeNCE's storing of arrays is that you can initialize one-dimensional arrays from *n*-dimensional arrays and vice-versa. For example, the following declaration is perfectly legal:

```
NODE 0
   NEW <> char a[10] = "abcdefghij";
   NEW <> char b[2][5] = a;
   NEW <> char c[2][5] = "abcdefghij";
   <> char a[0 : 4];
   null();
```

Here `b` and `c` will both equal `"abcde", "fghij"}`. Also, when HeNCE initializes such arrays, it makes them all point to separate chunks of memory. Therefore, changes to `a` do not affect `b`, even though the syntax might look that way. Moreover, changes to `a` do not affect `a[0 : 4]`, because they both point to different chunks of memory.

HeNCE attempts to perform type conversion as much as possible. Therefore, in the following declaration, `b` will be `97`, `98`, `99`; `c` will be `97.0`, `98.0`, `99.0`; and `e` will be `3.0`:

```
NODE 0
   NEW <> char a[3] = "abc";
   NEW <> int b[3] = a;
   NEW <> float c[3] = b;
   NEW <> int d = 3;
   NEW <> double e = d;
   null()
```

Subroutine call
8.5.1.2

Each node is bound to a subroutine. The subroutine call is where the arguments to the subroutine are defined and the return value is bound. The subroutine call is a part of the node program. The syntax of the subroutine call is as follows:

[*param* =] *sub-name* `(`[*expression* [,*expression*]*]`);`

The examples in Table 33 show how you use subroutine calls.

Table 33. Subroutine calls

Subroutine call	Description
`foo();`	Calls the subroutine `foo`.
`a = add(b, c);`	Calls the subroutine `add`, with the parameters `b` and `c` as arguments, and binds the return value to `a`.
`update_array(&(A[r][c]));`	Calls `update_array` with the address of element `(r,c)` of array `A`.

A parameter may be updated by subroutines in three ways:

- By binding it to the return parameter, for example, `a = foo().`

- By sending its address as an argument, for example, `foo(&a)`.
- By passing an array as an argument, for example, `foo(a[ ])`.

If the parameter to which the return value is bound has not been declared, it is output-only by default.

Output declarations
8.5.1.3

There may be some parameters that have been created or initialized by the subroutine call. These parameters can be declared as output-only after the subroutine call. Output declarations are of the following form:

```
> type param-name;
```

If node `1` wants to create the parameter `a` and initialize it to the result of `foo`, it can be declared as follows:

```
NODE 1
  a = foo();
  > int a;
```

A major difference between output and I/O parameters is in the assumptions about allocation: If an array is declared to be an I/O parameter, HeNCE allocates the space for it and initializes it from its ancestors, if any contain the array. However, if the array is declared as output-only, HeNCE assumes that the subroutine will allocate the array. Moreover, the size of the output array is determined after the subroutine call, as those determining parameters might be output-only as well.

Note

This behavior is not fully debugged in the current release of `htool`.

For example, suppose the subroutine `readtext` reads text from a file and returns the length in its first parameter, and the text in its second parameter. The HeNCE declaration for `readtext` should be as follows:

```
NODE 0
  readtext(&len, &txt);
  > int len;
  > char txt[len];
```

Output declarations can also be used for parameters that need to be created but are not used in the subroutine call. For example, suppose that the array `P` is going to be distributed among many nodes, but no one node is ever going to hold `P` in its entirety. `P` must be declared by some node, so that HeNCE knows its type and size; however, if it is declared as an input or input/output parameter , HeNCE allocates it and sets its values to null. If it is declared as an output parameter, HeNCE does not allocate it, but does note its type and size. Therefore, you can use output declarations to declare, but not allocate or use, an array.

Array parameters cannot change size during HeNCE's execution (unless a new parameter is created using `NEW`). For example:

```
NODE 0
  NEW <> int len = 400;
  <> char txt[len];
  readtext(&len, &txt);
```

If you make the preceding declaration and `readtext` changes `len` to `300`, `txt` will not undergo a size change; it will remain 400 characters long.

Special-purpose nodes 8.5.2

This subsection provides a description of the special-purpose nodes and their textual syntax as generated by HeNCE. As stated in subsection 8.4.2, page 101, special-purpose nodes all come in pairs surrounding a subgraph. In reality, special-purpose nodes are not nodes at all, but specify graph rewriting rules. The nature of the special-purpose node and the values of run-time variables determine how the subgraph gets written. For example, if a conditional node evaluates to false, the subgraph is not included in the graph's execution.

Loop nodes
8.5.2.1

Loop nodes are similar to fanout/fanin nodes, except that the copies of the subgraph are performed serially as shown in Figure 6.

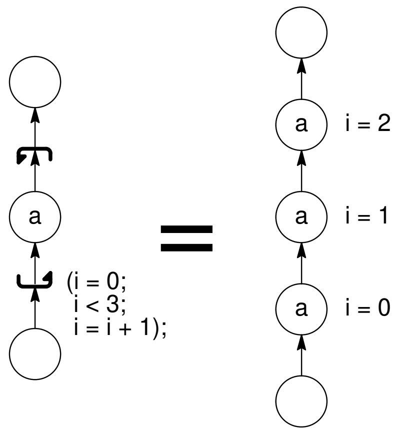

Figure 6. Loop expansion

The syntax is similar to the *for* syntax in C:

```
BEGINLOOP [x location y location] id
 ( [var = expression ] ;
 expression ;
 [var = expression ] );
ENDLOOP [x location y location] id;
```

When the loop is first encountered, an output integer parameter (*var*) is created and set to the first expression. If the middle expression evaluates to nonzero, the subgraph inside the loop is expanded and executed. When it finishes, *var* is updated using the last expression, and the second expression is rechecked. If the second expression evaluates to 0, execution continues from the ENDLOOP. Otherwise, the subgraph is expanded and reevaluated, and so on.

A current restriction enforced by HeNCE is that the *var* values in the first and third expression must be the same. If the *var* values are not the same, the parser returns an error.

Fanout and fanin nodes
8.5.2.2

Fanout nodes allow the user to dynamically create copies of a subgraph that will execute in parallel as shown in Figure 7. Fanin nodes mark the end of the subgraph to be spawned.

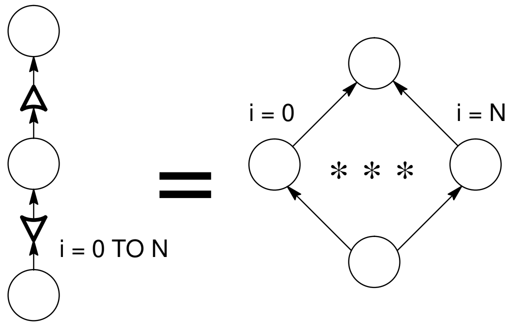

Figure 7. Fan expansion

Their syntax is as follows:

`FANOUT` [*x location y location*] `id;`
    var = *expression* `TO` *expression* ;
`FANIN` [*x location y location*] `id;`

The two expressions denote a range of values for *var*, which is treated as an output integer parameter. For each value of *var*, a separate copy of the subgraph is spawned. That copy of the subgraph may access its value of *var* as an integer parameter. If the first expression evaluates to a value less than the second, no copies are spawned; execution continues from the fanin node.

Pipe nodes
8.5.2.3

Pipe nodes allow control flow to be pipelined through a subgraph as shown in Figure 8.

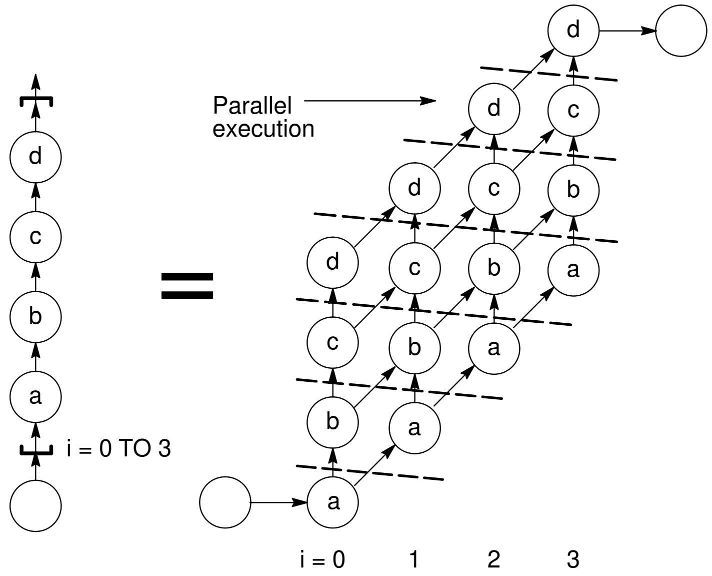

Figure 8. Pipe expansion

The syntax is similar to that used for fan nodes:

```
BEGINPIPE [x location y location] id
    var = expression TO expression ;
ENDPIPE [x location y location] id;
```

Again, for each value of *var*, a separate copy of the subgraph is spawned. However, now, for each node in the subgraph, there is an arc from its instance in one copy of the subgraph to its instance in the next copy of the subgraph (that is, from the instance where *var* = `i` to *var* = `i+1`).

Pipes allow for a more restricted type of parallelism than fanout/fanin nodes. Specifically, if node instances need to be serialized in a certain algorithm, the pipe construct can be effectively used.

Conditional nodes
8.5.2.4

Conditional nodes allow the user to choose whether or not subgraphs get executed, according to the run-time conditions of parameters as shown in Figure 9.

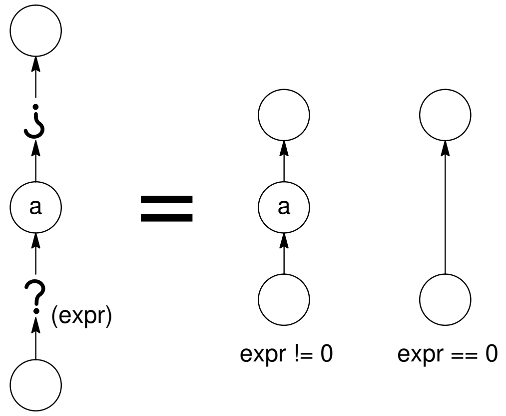

Figure 9. Conditional expansion

Like all other special-purpose nodes, conditionals come in begin/end pairs. The nodes between the pairs must form a subgraph. Otherwise, HeNCE flags an error before performing any execution.

The syntax of conditional nodes is as follows:

```
BEGINSWITCH [ x location y location ] id (expression );
ENDSWITCH [x location y location] id;
```

The expression is like any expression previously discussed. It may contain constants or parameters. All parameters are defaulted. As with C, if the expression evaluates to 0, it is treated as false, and the enclosed subgraph is ignored. If it evaluates to true, then the subgraph is executed normally.

Defining a cost matrix
8.6

A cost matrix does two things:

- Defines the list of host computer systems that make up the virtual parallel machine that will run the HeNCE program
- Provides HeNCE with an estimate of the cost of running each of the program's subroutines on each of the systems.

This helps the HeNCE system make intelligent choices about which system to use to run which subroutine.

Sequential execution
8.6.1

If you do not supply HeNCE with a cost matrix when you either build or run a program, it uses a default in which the nodes will run on a single system, the system on which `htool` is running. This means that no parallelism is realized since only one processor is available.

This mode is convenient when first learning HeNCE since you can run programs while completely ignoring the issue of cost matrices. It also may be helpful during program debugging.

Parallel execution
8.6.2

If you wish to exploit true parallel execution of your HeNCE program, you must allocate more than one system to your virtual parallel machine. To do this, you must define a cost matrix.

A cost matrix is a table, the rows of which are keyed by system names and the columns of which are keyed by subroutine names. There must be exactly one row in the cost matrix for each system in your virtual parallel machine, and there must be exactly one column for each subroutine called from your HeNCE graph. The enties in the cost matrix are integers that represent the relative cost of running a given subroutine on a given system as shown in Table 34.

Table 34. Example cost matrix

System name	`dprod` subroutine	`init` subroutine	`output` subroutine
star	4	2	6
moon	1	3	2

There are two systems in the virtual parallel machine. Their names are star and moon. A HeNCE program that uses this cost matrix runs using only these two systems. The HeNCE program contains three sequential subroutines that are called from nodes:

- `dprod`
- `init`
- `output`

The cost matrix shows that it is three times as expensive to run the `output` subroutine on star as it is on moon, but it is cheaper by a factor of two thirds on star. HeNCE uses this information to try to run subroutines on the systems on which they run cheapest.

The defintion of cost is arbitrary. HeNCE simply uses the numbers you provide. Most likely, you will choose to associate the costs with the subroutines' execution times on various systems. HeNCE provides no support for choosing costs. It does not measure the execution costs. You must estimate them somehow .

If you are running your program on a network of systems with similar performance characteristics (perhaps because all systems are the same kind and have similar loads), it is perfectly reasonable to give all costs in the matrix the same value. HeNCE will try to distribute the nodes on systems so that good parallelism is obtained.

The systems in the cost matrix can be of any system type for which you have installed PVM and the HeNCE libraries. HeNCE automatically compiles the subroutines for the systems on which it might execute them.

User interface
8.6.3

This section describes the mechanics of entering a cost matrix. Cost matrices are entered at the top of the PVM console area of the `htool` window in the scrolling region reserved for them. The `File` and `Edit` menu buttons in this area operate on cost matrices.

The easiest way to enter a cost matrix is to leave it blank until your HeNCE graph is completely drawn and all node programs are entered. At this point, HeNCE knows the names of all subroutines that are called from computation nodes. From the PVM Console Edit menu, select `Add from Graph` and a row for your first system will be added to the previously empty cost matrix. Fill in the system name and all costs for this row. Then pick `Add from Graph` to create the row for the second system, and so on.

When the matrix has been entered, save it into a file by picking `Save` from the PVM Console file menu. Do not forget to do this before exiting `htool`. The cost matrix is not automatically saved. By convention cost matrix files end with `.mat`, but this is not required.

To enter text into a field within a cost matrix, simply move the mouse to the field and type. The text will appear. Use the DELETE or BACKSPACE key to erase characters. Some other cost matrix manipulation functions follow:

1. To load a previously saved cost matrix, pick `Load` from the PVM Console File menu.
2. To create a blank new untitled and unsaved cost matrix, pick `New` from the PVM Console File menu. Your current cost matrix will be lost unless you save it first.
3. To add a row for a new system, pick `Add Host` from the PVM Console Edit menu.
4. To add a column for a new subroutine, pick `Add Subr` from the PVM Console Edit Menu.

Building and running a program 8.7

You use menu selections to build and run your program.

Building a program 8.7.1

Compile a HeNCE program by selecting `Build` from the PVM Console Build menu. Your program and subroutines are automatically compiled on all necessary system types as defined by the current cost matrix. If there are errors, messages appear in the Information segment of the `htool` window or perhaps in the center of the PVM Console segment.

Running a program 8.7.2

If you change either your HeNCE program, a sequential subroutine, or the current cost matrix, rebuild before attempting to run your program again. Run your program once it is compiled by selecting `Run Program` from the PVM Console Run menu.

Monitoring program execution 8.7.2.1

As the program runs, its progress is animated. The icons that make up the HeNCE graph change to show the node's state. Figure 10 defines the meaning of the changed icons.

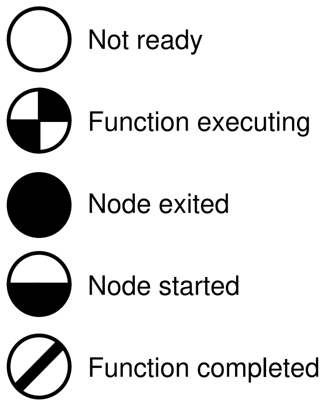

Figure 10. Meaning of run time and trace icons

The state of virtual parallel machine is displayed in the lower right of the `htool` window during the program run. There is a strip for each system. The meaning of the strips is color coded as shown in Table 35.

Table 35. Virtual parallel machine state display colors

Color display	Mono display	Meaning
Green	Dark gray	Node running
Yellow	Light gray	No node running
Gray	White	No task

Output from the program writes to the UNIX standard output (`stdout`) and appears in the scrolling region of the PVM Console window segment. Once your program begins running, it remains in run mode until you pick `Exit Run Mode` from the PVM Console Run menu. Until you do this, you cannot edit your HeNCE graph, and so on.

Using nodes
8.7.2.2

Each node in a HeNCE graph is a distinct process on some system. The arcs of the graph specify when a node is executed, which is when all of its parent nodes have finished executing. The nodes communicate with one another by using parameters, which are specified in the *node program*. Basically each node has three phases of its execution:

- Getting parameters: The first thing that a node does is read in its specified parameters. It first tries to get the parameters from its parent nodes. For example, if a node's parent specifies that it has parameter `B`, and the node needs parameter `B`, then the node sends a request for `B` to the parent. The parent responds to that request by sending the value of `B` to the node. If no parents have a parameter, the grandparents are checked, and so on.

  Certain types of parameters, such as arrays, can be split up among nodes. If a node wants all of an array, and its parents have split it up, then the node rebuilds the array from its parents. If the parents have just split up part of the array, then the node gets what it can from them, and tries to get the rest from its grandparents, and so on.

- Executing a subroutine: The computation of a node is performed by a C or Fortran subroutine. The arguments to the subroutine can be constants, parameters, or expressions involving parameters. The return value of the subroutine can likewise be bound to a parameter, and, using pointer indirection, the subroutine can modify parameters if they are properly given as arguments.

- Sending parameters: When the subroutine has executed, the node's execution is considered over, and its children can be started. The node cannot go away, however, as its children and other descendants may need its parameters. Thus, it stays to respond to requests for parameters. When it is sure that no more descendants will ask for parameters, it may go away.

Exiting `htool`
8.8

Trace information is written, by default, to a file called `hence.trace`. This can be changed by picking `Set Trace File` from the PVM Console Run menu. Trace data is always saved. As is described below, it can be used to replay the program's animation.

To run your program, the necessary PVM demons must also be running. If they are not, HeNCE starts them for you automatically.

Before exiting `htool`, you will probably want to halt them by picking `Halt PVM` from the PVM Console PVM menu. If you do not, you will likely leave unwanted processes running on all systems in your cost matrix, as well as the system that is running `htool`. It is quite possible that your HeNCE program will not run to completion due to bugs. For example, one of your sequential subroutines may contain a bug that causes a divide by zero or a segmentation violation. If this occurs, pick `Kill Program` from the PVM Console Run menu, followed by `Exit Run Mode`.

Tracing a program run
8.9

As a HeNCE program runs, it generates trace events. These events can be used to drive an animation of the HeNCE program either during or after its execution.

To view a program trace, enter the trace mode of `htool` by selecting the `Load Trace File` option under the `Trace` pull-down menu. The graph and cost matrix with which the program was run are displayed. The row of buttons just above the graph control the playback of the trace animation. Those buttons are as follows (listed left to right):

Button	Description
`rewind`	The first button is the `rewind` button (two left-facing open arrowheads). It rewinds the trace file to its beginning.
`reverse`	The second button (vertical bar to the right of an open, left-facing arrowhead) is the `reverse` step button. It causes the graph to single-step backwards in time.

Button	Description
`stop`	The third button is the `stop` button (the square symbol). It can be used to stop playback of the trace file.
`step`	The fourth button (vertical bar to the left of an open, right-facing arrowhead) is the `step` button. It can be used for single-stepping through the trace file.
`play`	The fifth button (a single, right-facing open arrowhead) is the `play` button, used to continuously play back the trace file.

During the trace, nodes in the program graph and the systems in the cost matrix are highlighted to indicate their state. The histogram at the lower right of the window also displays the state of each system over time. You can use the scrollbars on the histogram to display different parts of the histogram:

- Use the vertical scroll to see different systems.
- Use the horizontal scroll to move through time.
- Use the buttons (up and down arrows) in the histogram to change the time scale.

During tracing and execution, a red hair line indicates the current time.

Printing from HeNCE

8.10

It is possible to print the HeNCE graph. The `print` button in the `htool` window generates a PostScript version of the HeNCE graph. You can print this graph on your printer by using the command `lpr` *graph.ps* , where *graph.ps* is the file name you selected from the print window.

Tutorial Example [9]

This section describes the process of getting started, drawing nodes, drawing arcs, entering node programs, running a program, and quitting `htool`. The tutorial example is a simple HeNCE program that computes the dot product of two vectors in parallel. While you read this section, you can run `htool` and perform the steps as they are described.

The algorithm used in this tutorial is extremely simple. Figure 11 shows the parallel algorithm.

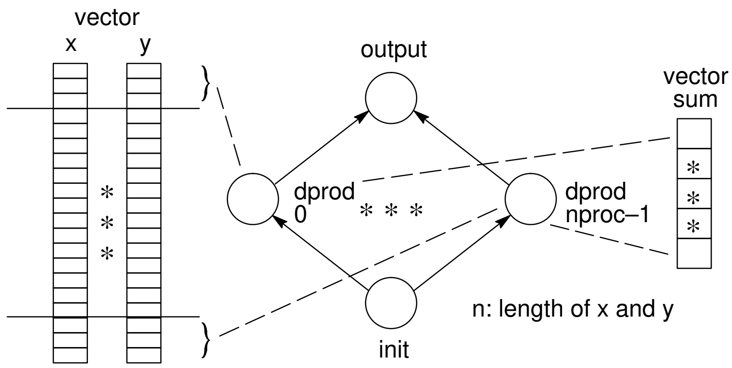

Figure 11. Parallel dot-product algorithm

Two vectors are allocated. The dot product is computed by splitting the vectors into `nproc` subvectors. The dot products of the subvectors are computed in parallel, and then the results are summed to form the dot product of the whole vectors.

A HeNCE fan construct is used to create the `dprod` nodes that run in parallel. The `init` node allocates storage for `x` and `y` and sets n to their length. It also gives the arrays their initial values. The `sum` vector holds the results of the dot product computation on each of the subvectors. Each replication of `dprod` accesses a subvector of `x` and `y` and performs a dot product on them. The result is stored in a single element of `sum`. The `output` node sums the entries in `sum` to form the final result.

Getting started
9.1

Before creating the HeNCE program, verify that you are running X windows (or perhaps Open Windows on a Sun) and that `htool` is in your command path.

Begin `htool` by entering `htool &` at the UNIX prompt so that runs it in the background. An `htool` window with no graph should appear. If the current language is not set to C, change it.

You can toggle the current language between C and Fortran by picking `Change Language` from the HeNCE menu at the upper left of the `htool` window. This menu also allows you to quit `htool`.

You can select the directory you want to use by selecting `Change Directory` from the HeNCE tool v2.0 menu.

Drawing nodes
9.2

The eventual goal is to draw a graph that looks like the one shown in Figure 12. You begin the graph by drawing the HeNCE graph nodes. Do not worry if your node placement is not exact or if the node numbers on your screen do not correspond to those shown in the figure, although they will if you draw the nodes in order from bottom to top.

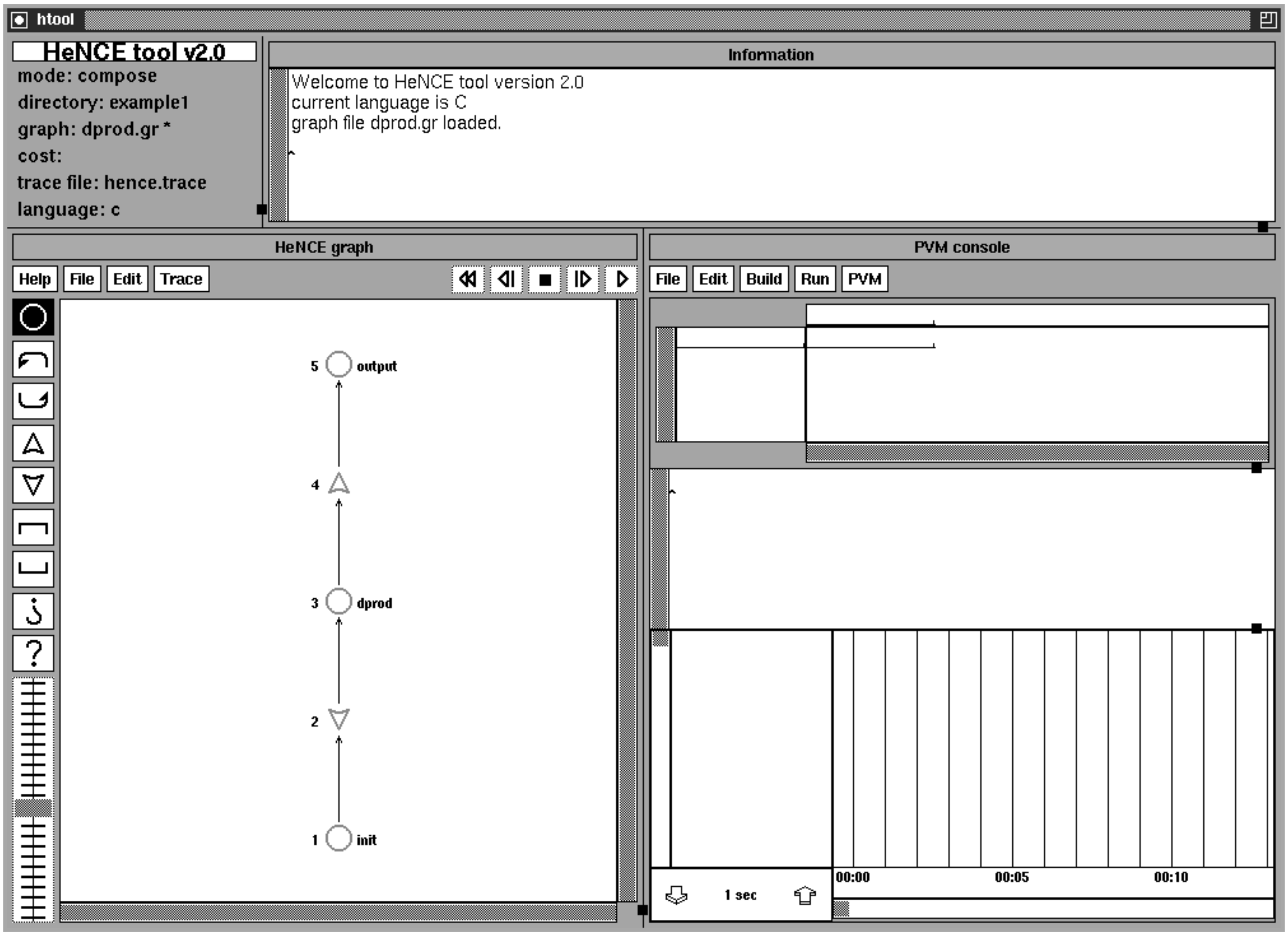

Figure 12. Dot-Product HeNCE Graph

1. Click on the circle icon on the left of the screen by pressing the left mouse button. This ensures that computation nodes are currently selected for drawing. (It is the default.)

2. Move the mouse to the lower center of the `htool` graph drawing area and click the left mouse button. A computation node should appear.

3. Draw the next node up from the bottom. Click (left) on the begin-fan node icon to select and then click (left) above the computation node just drawn. The begin-fan node should appear .

4. Continue this procedure for all of the nodes in the graph.

Drawing arcs
9.3

After the you have drawn the nodes, draw the four arcs in the graph. Notice that they all point up.

1. Move the mouse to the from node.
2. Press and hold the center mouse button.
3. Continue to hold the mouse button and move the mouse to the to node.
4. Release the mouse button.
5. Repeat the procedure for the remaining three arcs.

Entering node programs
9.4

At this point, you are ready to enter each node's program as well as inspect the node subroutines that have already been added.

1. Begin with the bottom node (labeled 1 in Figure 12). Open its node program by clicking the right mouse button on it. At this point, an editor window with text like that shown in Figure 13 appears.

```
NODE [ 307 116 ] 1
/* replace the UNDEFINED keyword with the node program */
/*
* Format: any input-only or input-output declarations
* NEW <> foo[100];
* followed by a subr call, e.g:
* x = xyzzy (a, b);
* then any output-only declarations:
* > bar[42];
*/
UNDEFINED;
```

Figure 13. Node program displayed for the new computation node

2. Edit the text to read like that shown in Figure 14, but remember not to change the first line. Replace the `UNDEFINED` line and (optionally) the comment with the desired node program.

```
NODE [ 307 116 ] 1
NEW <> int n=100;
NEW <> int nproc=10;
NEW <> float sum[nproc];
NEW <> float x[n];
NEW <> float y[n];
init(x, y, n);
```

Figure 14. Node program for the `init` node

   In this node program, the lines that begin with `NEW` declare variables and allocate storage for them. Many of the variables are also given initial values. The meaning within the program of the variables is described in Figure 11. The line containing `init` is the call to the `init` sequential subroutine, which is defined in the `init.c` file and gives initial values to the `x` and `y` vectors. When this node executes, `init` is called.

3. Close the editor window. The text you entered is parsed. If there is an error, `htool` complains and gives you a chance to revise. Otherwise, the subroutine name appears next to the node.

4. Examine the `init` subroutine by shift-clicking the right mouse button on the node. The subroutine appears in an editor window. If the `init.c` file does not already exist, type it in.

5. Inspect the node subroutines of the other computation nodes in the same way, after you have entered their node programs.

6. Give the begin-fan node (labeled 2) its node program as shown in Figure 15. Do not change the `FANOUT` line, but instead replace the rest with the `p =` line shown.

   The fan creates `nproc` copies of the subgraph (consisting of one computation node) within it. Each copy has access to a separate variable `p` that value that range from 0 to `nproc-1`, inclusive.

```
FANOUT [ 182 306 ] 2
p = 0 TO nproc-1;
```

Figure 15. Node program for the begin-fan node

7. Give the computation node (labeled 3) the node program shown in Figure 16.

```
NODE [ 174 247 ] 3
NEW < float
a[n/nproc]=x[p*(n/nproc):(p+1)*(n/nproc)-1];
NEW < float
b[n/nproc]=y[p*(n/nproc):(p+1)*(n/nproc)-1];
<> sum[p];
dprod(a, b, n/nproc, &sum[p]);
```

Figure 16. Node program for the `dproc` node

   This node is replicated by the fan. All copies of it run in parallel. The call to `dprod` computes the dot product of vectors `a` and `b`, which are subvectors of `x` and `y`. The lines with the word `NEW` define these subvectors.

   Notice that this program works only when `nproc` evenly divides `n`. More complicated expressions are needed for the general case.

The line <> `sum[p]` is interesting. The <> means that what follows is read from a predecessor node and that any changes made to it are available to successor nodes. So, each copy of this node uses of a single element `p` of the `sum` vector defined in the `init` node to store the result of its dot product. The elements of `sum` used by each copy of this `dproc` node are available as the `sum` vector to `output` node.

8. Skip the ending node (labeled 4) because a control construct ending node does not require a node program; hence, none is needed for the end-fan node.

9. Give the top computation node (labeled 5) the node program shown in Figure 17.

```
NODE [ 174 116 ] 5
< int nproc;
< float sum[];
output(sum, nproc);
```

Figure 17. Node program for the `outout` node

The < lines mean that this node gets `nproc` and `sum` from the nearest predecessor node that defines them. In this program, `nproc` comes from `init` and the elements of `sum` are set by the copies of the `dproc` nodes. The `output` subroutine totals the values in `sum` and prints the result.

Running the program
9.5

Having completed the program, you are ready to build and run it on the system on which you ran `htool`. This is the default if you do not enter a cost matrix.

1. Save the graph that you have drawn and annotated. Select `Save` from the File menu. Move the mouse over the file name box and enter text.

   Graph file names end in `.gr` by convention; so `dprod.gr` is a good name to choose.

2. Pick `Build` from the PVM Console Build menu. Watch the `htool` window for error messages. If there are none, you can run your program.

The `.o` files are saved in your current working directory. The executable files are saved in directory `$(HOME)/pvm3/bin/$(PVM_ARCH)`. `PVM_ARCH` is the environment variable that defines the name of the system architecture on which you are running.

3. Select `Run Program` from the PVM Console Run menu. You should see animation on the HeNCE graph as the program runs.

   When the program completes, the following message appears on the PVM console.

   ```
   HeNCE executioner exited normally
   ```

4. Pick `Exit Run Mode` from the PVM Console Run menu. HeNCE has created a cost matrix for you. It contains only one system, the one on which you are running `htool`.

When a HeNCE program runs, it creates a trace file which you can use to review your programs execution.

1. Pick `Load Trace File` from the HeNCE Graph Trace menu. Notice that HeNCE has created a trace file with the default name `hence.trace`.

2. Load it to enter trace mode. The VCR-like control icons in the upper right of the graph segment control the replay.

   a. Click the `play` button. You will see the run-time animation repeated.

   b. Press the left-most button to rewind.

   c. Press the second button from the right repeatedly. This is the single step forward button.

   d. Press the second button to take a single step backwards.

   e. Press the middle button is stop.

3. Select `Exit Trace Mode` from the Trace menu when you finish experimenting with trace replay.

Running on two systems 9.6

You can run the dot-product program using two systems. To do this, you must add another system to the cost matrix. That is, a system that you can access with the `rlogin`(1) command without entering a password and on which PVM and the HeNCE libraries are installed.

1. Pick `Add from Graph` from the PVM Console Edit window. Another row is added to the default cost matrix created in the previous step.

2. Enter values in all the fields by moving the mouse to a field and typing. Enter the host name of the second system and costs for all subroutines. You can use the value 1 for all costs.

3. Save the cost matrix by picking `Save` from the PVM Console File menu and enter a file name. By convention, this name ends in `.mat`.

4. Pick `Build` from the PVM Console Build menu. Watch the `htool` window for error messages. If there are none, you can run your program.

   The `.o` files are saved in your current working directory. The executable files are saved in directory `$(HOME)/pvm3/$(PVM_ARCH)`. `PVM_ARCH` is the environment variable that defines the name of the system architecture on which you are running.

5. Select `Run Program` from the PVM Console Run menu. You should see animation on the HeNCE graph as the program runs.

   When the program completes, the following message appears on the PVM console.

   ```
   HeNCE executioner exited normally
   ```

6. Pick `Exit Run Mode` from the PVM Console Run menu. HeNCE has created a cost matrix for you. It contains only one system, the one on which you are running `htool`.

Quitting `htool`
9.7

You have completed the tutorial. You have entered and run a parallel program using more than one system, and you have replayed its execution using the HeNCE trace facility.

1. Select `Halt PVM` from the PVM Console PVM menu. Do not forget to do this, or you will leave PVM daemon processes running.

2. Select `Quit` from the HeNCE Menu at the upper left of the htool window to exit `htool`.

PVM Error Messages [A]

Table 36 lists the errors detected by PVM. These include the following:

- 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 Cray Research extensions.

Table 36. Error messages issued by PVM functions

Error Text	Value	Symbol	Other Information
	0	`PvmOk`	
	–1		Reserved
`Bad parameter`	–2	`PvmBadParam`	
`Count mismatch`	–3	`PvmMismatch`	
	–4		Reserved
`End of buffer`	–5	`PvmNoData`	
`No such host`	–6	`PvmNoHost`	
`No such file`	–7	`PvmNoFile`	
	–8		Reserved

Table 36. Error messages issued by PVM functions
(continued)

Error Text	Value	Symbol	Other Information
	–9		Reserved
`Malloc failed`	–10	`PvmNoMem`	
	–11		Reserved
`Can't decode message`	–12	`PvmBadMsg`	
	–13		Reserved
`System error`	–14	`PvmSysErr`	
`No current buffer`	–15	`PvmNoBuf`	
`No such buffer`	–16	`PvmNoSuchBuf`	
`Null group name`	–17	`PvmNullGroup`	
`Already in group`	–18	`PvmDupGroup`	
`No such group`	–19	`PvmNoGroup`	
`Not in group`	–20	`PvmNotInGroup`	
`No such instance`	–21	`PvmNoInst`	
`Host failed`	–22	`PvmHostFail`	
`No parent task`	–23	`PvmNoParent`	
`Not implemented`	–24	`PvmNotImpl`	
`Pvmd system error`	–25	`PvmDSysErr`	
`Version mismatch`	–26	`PvmBadVersion`	
`Out of resources`	–27	`PvmOutofRes`	
`Duplicate host`	–28	`PvmDupHost`	
`Can't start pvmd`	–29	`PvmCantStart`	
`Already in progress`	–30	`PvmAlready`	
`No such task`	–31	`PvmNoTask`	

Table 36. Error messages issued by PVM functions
(continued)

Error Text	Value	Symbol	Other Information
`No such entry`	–32	`PvmNoEntry`	
`Duplicate entry`	–33	`PvmDupEntry`	
`Name too long`	–100	`PvmTooLong`	Name exceeds 32-character limit
`Async xfers still active`	–101	`PvmStillActive`	
`Precision lost on DataDefault pack`	–102	`PvmLostPrecision`	Precision lost on default packing of 64-bit integers
`Out of resources--data buffers`	–103	`PvmOutOfResBuf`	See subsection 2.4.2, page 24.
`Out of resources--shared memory pool`	–104	`PvmOutOfResSMP`	See subsection 2.4.1, page 23.
`Out of resources--group members`	–105	`PvmOutOfResGmems`	See subsection 2.4.3, page 25.
`Too much data packed`	–106	`PvmTooMuchData`	See subsection 2.3.3.4, page 20.
`Total pack memory limit exceeded`	–107	`PvmMemLimit`	See subsection 2.3.3.5, page 20.

Comparison of Message-passing Libraries [B]

This appendix compares the message-passing library functions of several different MPP systems with the functions in PVM that are supported on CRAY T3D systems. It demonstrates that PVM provides the basic functionality available with the other libraries and that conversion of applications that use these other libraries is fairly straightforward.

Each subsection covers the interface from a specific library. The libraries covered are:

- nCUBE (release 2)
- Intel iPSC/2 and iPSC/860

This appendix focuses on PVM being used only for communication between nodes in a CRAY T3D partition; it does not specifically include communication outside the partition (although the code conversions would be very similar).

This appendix refers to PVM functions (and the functions for other libraries) by using their C language interface names. PVM (and the other libraries) also support Fortran language interfaces to all functions.

Note

This appendix focuses on changes using portable PVM syntax. Cray Research has added extensions to PVM (PVM channels) that may provide better performance and that you may want to consider using instead.

---

nCUBE release 2
B.1

The nCUBE message-passing functions use a read/write model that is similar to I/O, but slightly different from the more common send/receive model used by PVM.

Writing a message B.1.1

The `nwrite` function queues a message for transfer. It has the following syntax:

```
int nwrite (char *buffer , int nbytes , int dest ,
   int type , int *flag )
```

To use PVM, code the following:

```
pvm_psend (instance , type , buffer , nbytes , PVM_BYTE)
```

The *instance* argument should be the same as the relative node ID portion of the nCUBE *dest* argument. The nCUBE 2 *flag* argument is defined as currently unused; in PVM it is not important.

With `nwrite`, the buffer may be reused upon return from the `nwrite` call. With PVM, the buffer may be reused upon return from the `pvm_psend`(3) call.

Copying a message from a process buffer B.1.2

The `nread` function reads a message into user space to a specified address. It has the following syntax:

```
int nread (char *buffer , int nbytes , int *source , int
   *type , int *flag );
```

Code the following for PVM:

```
pvm_precv (*source , *type , buffer , nbytes , PVM_BYTE,
     source, type,   nread-return-value);
```

Both `pvm_precv`(3) and `nread` allow *source* and *type* to contain a value of –1 as a wildcard. In PVM, one argument is used to pass the selection value, and a second is used to return the selection value.

The *source* argument returns a pvm_tid, not a PE number.

Both nread and pvm_precv block if no message has arrived that meets your request.

Testing for a message
B.1.3

The ntest function tests for a message in the message pool. It has the following syntax:

```
int ntest (int *source , int *type )
```

The following should be coded:

```
ibuf = pvm_nrecv (*source , *type );
```

If *ibuf* is greater than 0, a message is selected, and you can call pvm_bufinfo(3) to obtain information on its length, message type, or sender.

The pvm_nrecv(3) function differs from ntest in that, with pvm_nrecv, the message has actually been received. With ntest, it has not; nread must be called to actually receive the message. With PVM, this call to nread should be replaced by a call to pvm_upkbyte(3).

Example:

```
if (ntest (&otherpe, &msgtag)) {
    nread(buf,sizeof(buf),&otherpe,&msgtag,&flag);
    ...
}
```

This can be replaced by the following:

```
    if (pvm_nrecv (otherpe,msgtag) > 0) {
        pvm_upkbyte (buf,sizeof(buf));
    }
```

Intel iPSC/2 and iPSC/860
B.2

The Intel message-passing library uses a send/receive model similar to that in PVM. The library provides some additional capabilities that are not directly supported by PVM at this time.

Sending a message
B.2.1

The Intel library supports the following three ways to send a message:

```
csend(long type, char *buf, long len, long node,
  long pid)

hsend(long type, char *buf, long len, long node,
  long pid, void (* proc)())

long isend(long type, char *buf, long len,
  long node, long pid)
```

The `csend` function is a synchronous, blocking request, while `hsend` and `isend` are asynchronous requests. With `csend`, the *buf* argument may be reused as soon as `csend` returns; with `isend`, the *buf* argument may not be reused until the code has verified that the message has been received or has canceled the message. With `hsend`, the *buf* argument may not be reused until the *proc* handler has been called.

The `pvm_psend`(3) function falls between these cases. With the `snd` function, the buffer may be reused as soon as `pvm_psend` returns (as with `csend`), but there is no guarantee that the message has been received (as with `isend` and `hsend`). Each application has to assess the impact of this difference. In general, however, the following should be coded to send a message in PVM:

```
pvm_psend(node, type, buf, len, PVM_BYTE)
```

The Intel *pid* argument refers to the process ID (0 for an RX system) and is not required in PVM. When PVM is used to communicate with the CRAY Y-MP system or another CRAY T3D partition, the pvm_tid includes information roughly corresponding to the Intel *pid* and *node* arguments. The *node* argument refers to the processing element in both the Intel and PVM libraries.

The Intel library uses the special value of –1 to mean broadcast to all nodes. If this is being used, the following should be coded in PVM:

```
pvm_initsend();

pvm_pkbyte(buf, len, 1);

pvm_bcast(0, type);
```

The Intel library provides a special meaning for other negative values, specifying transmission to a surrounding subcube. This is not supported by PVM.

Receiving a message B.2.2

The Intel library supports the following three ways to receive a message:

```
crecv(long typesel, char *buf, long len);

hrecv(long typesel, char *buf, long len,
   void(* proc)());

long irecv(long typesel, char *buf, long len);
```

The `crecv` function is a blocking receive request that corresponds directly to the `pvm_recv` call. `hrecv` is an asynchronous receive request for which the *proc* handler function is called when the message arrives. `irecv` is an asynchronous request that returns immediately but requires the user application to subsequently check for arrival of the message. Most of the time the `pvm_nrecv` call can replace `irecv`.

To convert a call to `crecv`, code the following:

```
pvm_precv(-1, typesel , buf, len , PVM_BYTE,
   0, 0, 0);
```

Both PVM and Intel interpret a *typesel* value of –1 to mean "receive a message of any type." Intel uses other negative numbers as a mask defining a set of message types; this is not directly supported by PVM. (`pvm_recvf` can be used to obtain similar functionality if necessary. In this case, `pvm_recv` and `pvm_upkbyte` must be used instead of `pvm_precv`.)

There are no PVM features that correspond to `hrecv`. Applications that use this function must be analyzed to assess the best way to convert them to use PVM.

PVM does not have a feature that directly corresponds to `irecv`. In most cases, however, you can use `pvm_nrecv` to obtain similar functionality. `pvm_nrecv` is nothing more than a nonblocking receive request. The main thing to remember is that, if one call fails, you must issue the same request again; PVM does not "remember" the earlier request. So, if you are using `irecv` simply as a nonblocking request, you can usually replace it and the subsequent calls to `msgdone` by calls to `pvm_nrecv`.

Example:

```
id = irecv (typesel , buf , len );
while (! msgdone (id)) {
/* do stuff that does not depend on message */
}
```

This code can be converted to the following:

```
while (! pvm_nrecv (-1, typesel ))) {
/* do stuff that does not depend on message */
}
pvm_upkbyte (buf , len , 1);
```

Your application may be using `irecv` to overlap communication and computation. On a CRAY T3D system, there is little value in doing this. You should replace `irecv` by calls to either `pvm_recv` (blocking) or `pvm_precv` (nonblocking).

Sending a message and waiting for a response
B.2.3

The Intel library supports three ways to issue a single request to send a message and wait for a response. The calling sequences for these functions (`csendrecv`, `hsendrecv`, and `isendrecv`) are not shown here, but are basically those resulting from combining the calling sequences for the corresponding send and receive calls. PVM does not provide combined functions in the same manner; conversion should follow the previous suggestions and discussion for the individual send and receive functions.

Determining if a message is pending
B.2.4

The Intel library supports two ways to see if a message has arrived and is pending. These are as follows:

```
cprobe (long typesel );

long iprobe (long typesel );
```

The `cprobe` function is a blocking call and `iprobe` is an asynchronous call. PVM has a single `pvm_probe`(3) function that corresponds to the Intel `iprobe` function. A blocking version is not available.

A probe is most often used in two different ways, and, in both cases, it can be replaced by calls to `pvm_recv` or `pvm_nrecv`.

In one case, it is used as a nonblocking check to see if a message has arrived. If it is used in this way, it is immediately followed by a receive request. This syntax can be easily replaced by calls to `pvm_nrecv`, followed (if the message arrived) by a call to `pvm_upkbyte`.

In the other case, a message is probed for, and then one or more of the information functions is called to learn more about the message. For example, `infocount` may be called, sufficient memory allocated, and then a receive function called to get the data. Or `infotype` may be called, and, depending on the message type, a particular buffer may be used or another function may be called to process the message. Because PVM separates the concepts of receiving the message and receiving the data, it is usually easy to convert codes written in this style.

Example:

```
cprobe (typesel);
len = infocount ();
buf = malloc (len);
crecv (typesel, buf, len);
```

This can be replaced by the following:

```
bufid = pvm_recv (-1, typesel);
pvm_bufinfo (bufid, &len, 0, 0);
buf = malloc (len);
pvm_upkbyte (buf, len, 1);
```

Obtaining pending or received message status B.2.5

The Intel library has a set of four functions that return information about a pending or received message. These functions have the following syntax:

```
long infocount ()

long infonode ()

long infopid ()

long infotype ()
```

If you use `pvm_precv`, this information is available on return, through arguments passed to `pvm_precv`. If you use `pvm_recv`, `pvm_nrecv`, or `pvm_probe`, the return code is a *buffer identifier*, which you can later use to obtain this information. To do this, call `pvm_bufinfo`, which has the following syntax:

```
int pvm_bufinfo(int buf, int *bytes,
   int *msgtag, char *pvm_tid);
```

The correspondence between the two sets of functions is as follows.

- The Intel `infocount` function corresponds to the `pvm_bufinfo` *bytes* argument.
- The Intel `infonode` function corresponds to the `pvm_bufinfo` *pvm_tid* argument.
- The Intel `infopid` function (which always returns 0 for RX systems) does not apply with PVM.
- The Intel `infotype` function corresponds to the `pvm_bufinfo` *msgtag* argument.

The `pvm_bufinfo` *buf* argument is the value returned by the `pvm_recv`, `pvm_nrecv`, and `pvm_probe` functions.

The code conversions for these cases should be obvious. The `pvm_bufinfo` function need not be called immediately after the message is received (unlike the Intel calls), and it refers to the message identified by *buf*, which need not be the most recently received message.

Managing an asynchronous request B.2.6

Because the Intel library supports several styles of asynchronous requests, a set of functions lets the application subsequently manage these requests. These functions (`flushmsg`, `masktrap`, `msgcancel`, `msgdone`, and `msgwait`) have no direct counterparts in PVM. As discussed earlier, applications that use these functions have to be assessed for the best conversion approach.

Differences Between PVM Version 2 and Version 3 [C]

This appendix describes the functional interfaces for PVM version 2.*x*. Some older applications may have been written using this older version of PVM and must be converted. Version 2.*x* of PVM provides a subset of the functionality available with PVM version 3.*x*. The main purpose of this appendix is to indicate how existing code using the version 2.*x* interfaces should be changed to use the new interfaces.

A major difference between version 2.*x* and version 3.*x* is a move from using a component/instance pair to using an integer pvm_tid to identify PVM tasks. The component/instance pair supplied two functions: identification of individual tasks and the formation of groups of related tasks (a task was limited to membership in a single group identified by the component name). With PVM version 3.*x*, these two functions have been separated as follows:

- An integer value, the pvm_tid, is used to identify tasks.
- Tasks can dynamically join and leave groups, which are identified by a character string.
- Members of groups can be referred to by using a group-name/instance pair, very similar to the version 2.*x* component/instance pair; a conversion function (`pvm_gettid`(3)) allows the pvm_tid for a particular pair to be obtained.

Existing PVM version 2.*x* code will have to be evaluated to see whether it is worthwhile to retain the name/instance pair or to use the simpler pvm_tid.

Task control functions

C.1

Task control functions for PVM version 2.*x* and version 3.*x* are shown in Table 37.

Table 37. Task control function cross-reference

PVM function		Remarks
Version 2. *x*	Version 3.*x*	
`int initiate` `int initiateM`	`pvm_spawn`	
`int enroll`	`pvm_mytid` `pvm_joingroup`	There are two ways this function may have been used. If used simply to enroll the task with PVM and to obtain an instance number, `pvm_mytid`(3) can be used. If used to form a static group with the name of component, use the `pvm_joingroup`(3) function.
`void leave`	`pvm_exit`	
`int status`	`pvm_pstat`	
`int terminate`	`pvm_kill`	
`int whoami`	`pvm_mytid`	This function has been replaced by `pvm_mytid`. The `whoami` function also returned the component name; there is no function to determine all the groups containing a task.
`int pstatus`	`pvm_config`	

Data transmittal functions C.2

Data transmittal functions for PVM version 2.*x* and version 3.*x* are shown in Table 38.

Table 38. Data transmittal function cross-reference

PVM function		Remarks
Version 2. *x*	Version 3.*x*	
`void initsend`	`pvm_initsend`	

`int putnint`
`int putnshort`
`int putnlong`
`int putnfloat`
`int putndfloat`
`int putncplx`
`int putndcplx`
`int putnbytes`
`int putstring`
	`pvm_pkint`
	`pvm_pkshort`
	`pvm_pklong`
	`pvm_pkfloat`
	`pvm_pkdouble`
	`pvm_pkdcplx`
	`pvm_pkcplx`
	`pvm_pkbyte`
	`pvm_pkstring`
		These functions have all been replaced by corresponding functions beginning with the string `pvm_pk`. The version 2.*x* functions supported only a unit stride; most of the version 3.*x* functions support an arbitrary stride.

`int snd`
`int vsnd`
	`pvm_send`
	`pvm_bcast`
	`pvm_mcast`
		These functions have been replaced by the `pvm_send`(3), `pvm_bcast`(3), and `pvm_mcast`(3) functions. Broadcasting is now done by an explicit broadcast function, and not implicitly by a special task identifier. The "virtual" channels in release 2.*x* (`vsnd`) are supported through the `PvmRoute` option in the `pvm_setopt`(3) function.

Data receipt functions
C.3

Data receipt functions for PVM version 2.*x* and version 3.*x* are shown in Table 39.

Table 39. Data receipt function cross-reference

PVM function		Remarks
Version 2.*x*	Version 3.*x*	

`int rcv`
`int rcvmulti`
`int vrcv`
`int vrcvmulti`
	`pvm_recv`	These functions have been replaced by `pvm_recv`(3), which also supports selection by sender semantics. There is no directly corresponding function for `rcvmulti` and `vrcvmulti`. If desired, this functionality can be obtained using the `pvm_recvf`(3) function. The 2.*x* "virtual" channels (`vrecv` and `vrecvmulti`) are supported through the `PvmRoute` option of `pvm_setopt`(3). The version 3.*x* functions return a receive buffer; you can call `pvm_bufinfo`(3) to obtain the message type.

`int probe` `int probemulti`	`pvm_probe`	`probemulti` functionality is not directly supported.
`int rcvinfo`	`pvm_bufinfo`	The name has been changed to note that the function is examining any message buffer .

`int getnint`
`int getnshort`
`int getnlong`
`int getnfloat`
`int getndfloat`
`int getncplx`
`int getndcplx`
`int getnbytes`
`int getstring`
	`pvm_upkint`
	`pvm_upkshort`
	`pvm_upklong`
	`pvm_upkfloat`
	`pvm_upkdouble`
	`pvm_upkcplx`
	`pvm_upkdcplx`
	`pvm_upkbytes`
	`pvm_upkstring`
		These functions have all been replaced by corresponding functions beginning with the string `pvm_upk`. The version 2.*x* functions supported only a unit stride; most of the version 3.*x* functions support an arbitrary stride.

Task synchronization functions C.4

Task synchronization functions for PVM version 2.*x* and version 3.*x* are shown in Table 40.

Table 40. Task synchronization function cross-reference

PVM function		Remarks
Version 2.*x*	Version 3.*x*	
`int barrier`	`pvm_barrier`	The name in `pvm_barrier`(3) is a group name, not a barrier name.
`int ready` `int waituntil`		Not supported. On the CRAY T3D system, events are supported in `libu`.

PVM Man Pages [D]

This appendix contains copies of the PVM man pages, which document the following commands and functions (arranged alphabetically).

man1

- htool(1)
- pvm(1)
- pvm_intro(1)
- pvmd3(1)

man3

- 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_gather(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.

fanin nodes
: Fanin nodes mark the end of the subgraph to be spawned.

fanout nodes
: Fanout nodes allow the user to dynamically create copies of a subgraph that will execute in parallel.

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

HeNCE
: Heterogeneous Network Computing Environment.

loop nodes
: Loop nodes are similar to fanout/fanin nodes, except that the copies of the subgraph are performed serially.

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* .

normal nodes Normal nodes are used to indicate computation and have node subroutines (implemented in C or Fortran) associated with them that perform the user's computation.

nonblocking A nonblocking function is one that returns immediately.

ORNL Oak Ridge National Laboratory.

PE Processing element.

pipe nodes Pipe nodes allow control flow to be pipelined through a subgraph.

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.

special nodes Special nodes are used in pairs to delimit subgraphs of the HeNCE graph; they allow a portion of the graph to be repeated conditionally , sequentially, or in parallel during execution of the program.

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.

XDR eXternal Data Representation.

Index

E

F

K

L

M

N

O

P

Q

R

S

T

U

V

W

X