Parallel Environment for AIX and Linux

Operation and Use

Version 5 Release 1

SC23-6667-00

Parallel Environment for AIX and Linux

Operation and Use

Version 5 Release 1

SC23-6667-00

Note

Before using this information and the product it supports, read the information in “Notices” on page 245.

First Edition (November 2008)

This edition applies to version 5, release 1, modification 0 of IBM Parallel Environment for AIX (product number 5765-PEA) and version 5, release 1, modification 0 of IBM Parallel Environment for Linux (product number 5765-PEL), and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. A form for readers’ comments may be provided at the back of this publication, or you can send your comments to the following address:

International Business Machines Corporation

Department 58HA, Mail Station P181

2455 South Road

Poughkeepsie, NY 12601-5400

United States of America

FAX (United States & Canada): 1+845+432-9405

FAX (Other Countries):

Your International Access Code +1+845+432-9405

IBMLink™ (United States customers only): IBMUSM10(MHVRCFS)

Internet e-mail: mhvrcfs@us.ibm.com

If you want a reply, be sure to include your name, address, and telephone or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this publication
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Tables

About this information

This information describes the IBM® Parallel Environment (PE) program product and its Parallel Operating Environment (POE). It shows how to use POE's facilities to compile, execute, and analyze parallel programs.

This information concentrates on the command line tasks associated with POE, as opposed to the writing of parallel programs. For this reason, you should use this information in conjunction with *IBM Parallel Environment: MPI Subroutine Reference* and *IBM Parallel Environment: MPI Programming Guide*.

For complete information on installing the PE software and setting up users, see *IBM Parallel Environment: Installation*.

This information supports the following program products:
- IBM Parallel Environment for AIX® (5765-PEA), Version 5 Release 1 Modification 0
- IBM Parallel Environment for Linux® (5765-PEL) Version 5 Release 1 Modification 0

To make this information easier to read, the name *IBM Parallel Environment* has been abbreviated to *PE for AIX*, *PE for Linux*, or more generally, PE throughout.

For AIX users:

The PE for AIX information assumes that one of the following is already installed:
- AIX Version 5.3 Technology Level 5300-09 (AIX V5.3 TL 5300-09)
- AIX Version 6.1 (or later), either standalone or connected by way of an Ethernet LAN supporting IP.

For information on installing AIX® see the *AIX Installation Guide and Reference*.

Note: AIX Version 5.3 Technology Level 5300-09 (or *AIX V5.3 TL 5300-09*) identifies the specific AIX 5.3 maintenance level that is required to run PE 5.1.0. The name AIX 5.3 is used in more general discussions.

For Linux users:

The PE for Linux information assumes that one of the following Linux distributions is already installed:
- SUSE LINUX Enterprise Server (SLES) 10
- Red Hat Enterprise Linux 5, Update 2

Who should read this information

This information is designed primarily for end users and application developers. It is also intended for those who run parallel programs, and some of the information covered should interest system administrators. Readers should have knowledge of the AIX or Linux operating system. Where necessary, background information relating to these areas is provided. More commonly, you are referred to the appropriate documentation.

How this information is organized

Overview of contents

This information is organized as follows:

- *Introduction* is a quick overview of the PE program product. It describes the various PE components, and how you might use each in developing a parallel application program.
- *Executing parallel programs* describes how to compile and execute parallel programs using the Parallel Operating Environment (POE).
- *Managing POE jobs* includes information on allocating nodes with Tivoli® Workload Scheduler LoadLeveler® (LoadLeveler), and the environment variables to use when running your applications.
- *Using the Distributed Interactive Shell (DISH)* describes DISH, an interactive tool that can be used as a control center to multiple distributed copies of a client, and can be configured into a distributed shell, a parallel debugger, or some other interactive program.
- *Parallel Environment commands* contains the manual pages for the PE commands discussed throughout this information.
- *POE environment variables and command line flags* describes the environment variables you can set to influence the execution of parallel programs and the operation of PE tools. This appendix also describes the command line flags associated with each of the environment variables. When invoking a parallel program, you can use these flags to override the value of an environment variable.
- *Profiling programs with the gprof command* provides a brief explanation of how to profile a serial or parallel program using the **gprof** command.
- *Dish commands* contains the manual pages for DISH-related commands and subcommands.

Conventions and terminology used in this information

Note that in this information, LoadLeveler is also referred to as *Tivoli® Workload Scheduler LoadLeveler* and *TWS LoadLeveler*.

This information uses the following typographic conventions:

Table 1. Typographic conventions

Convention	Usage
bold	**Bold** words or characters represent system elements that you must use literally, such as: command names, file names, flag names, path names, PE component names (**poe**, for example), and subroutines.
`constant width`	Examples and information that the system displays appear in `constant-width` typeface.
italic	*Italicized* words or characters represent variable values that you must supply. *Italics* are also used for book titles, for the first use of a glossary term, and for general emphasis in text.
[item]	Used to indicate optional items.
<Key>	Used to indicate keys you press.

Table 1. Typographic conventions (continued)

Convention	Usage
\	The continuation character is used in coding examples in this information for formatting purposes.

In addition to the highlighting conventions, this information uses the following conventions when describing how to perform tasks.

User actions appear in uppercase boldface type. For example, if the action is to enter the **tool** command, this information presents the instruction as:

ENTER
tool

Abbreviated names

Some of the abbreviated names used in this information follow.

AIX	Advanced Interactive Executive
CSM	Clusters Systems Management
CSS	communication subsystem
CTSEC	cluster-based security
dsh	distributed shell
GUI	graphical user interface
HDF	Hierarchical Data Format
IP	Internet Protocol
LAPI	Low-level Application Programming Interface
MPI	Message Passing Interface
PE	IBM Parallel Environment for AIX or IBM Parallel Environment for Linux
PE MPI	IBM's implementation of the MPI standard for PE
PE MPI-IO	IBM's implementation of MPI I/O for PE
POE	parallel operating environment
RSCT	Reliable Scalable Cluster Technology
rsh	remote shell
STDERR	standard error
STDIN	standard input
STDOUT	standard output
System x™	IBM System x

Prerequisite and related information

The Parallel Environment library consists of:

- IBM Parallel Environment: Installation, SC23-6666
- IBM Parallel Environment: Operation and Use, SC23-6667
- IBM Parallel Environment: Messages, SC23-6669

- IBM Parallel Environment: MPI Programming Guide, SC23-6670
- IBM Parallel Environment: MPI Subroutine Reference, SC23-6671

To access the most recent Parallel Environment documentation in PDF and HTML format, refer to the IBM Clusters Information Center, on the Web.

Both the current Parallel Environment books and earlier versions of the library are also available in PDF format from the IBM Publications Center on the Web.

It is easiest to locate a book in the IBM Publications Center by supplying the book's publication number. The publication number for each of the Parallel Environment books is listed after the book title in the preceding list.

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have comments about this information or other PE documentation:

- Send your comments by e-mail to: mhvrcfs@us.ibm.com

  Be sure to include the name of the book, the part number of the book, the version of PE, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).
- Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative.

National language support (NLS)

For national language support (NLS), all PE components and tools display messages that are located in externalized message catalogs. English versions of the message catalogs are shipped with the PE licensed program, but your site may be using its own translated message catalogs. The PE components use the environment variable **NLSPATH** to find the appropriate message catalog. **NLSPATH** specifies a list of directories to search for message catalogs. The directories are searched, in the order listed, to locate the message catalog. In resolving the path to the message catalog, **NLSPATH** is affected by the values of the environment variables **LC_MESSAGES** and **LANG**. If you get an error saying that a message catalog is not found and you want the default message catalog, do the following.

Table 2. Specifying the default message catalog with the NLSPATH environment variable

If you are using PE for AIX:	If you are using PE for Linux:
ENTER **export NLSPATH=/usr/lib/nls/msg/%L/%N** **export LANG=C**	**ENTER** **export NLSPATH=/usr/share/locale/%L/%N** **export LANG=en_US**

The PE message catalogs are in English, and are located in the following directories.

Table 3. Location of PE message catalogs

If you are using PE for AIX:	If you are using PE for Linux:
/usr/lib/nls/msg/C	**/usr/share/locale/C**
/usr/lib/nls/msg/En_US	**/usr/share/locale/En_US**
/usr/lib/nls/msg/en_US	**/usr/share/locale/en_US**
	/usr/share/locale/en_US.UTF-8

If your site is using its own translations of the message catalogs, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**.

PE for AIX users can refer to *AIX: General Programming Concepts: Writing and Debugging Programs* for more information on NLS and message catalogs.

Functional restrictions for PE 5.1

Functional restrictions for PE for AIX 5.1:

- Because PE Version 5 Release 1 exploits the barrier synchronization register (BSR), any user attempting a read-modify-write operation on MPI library allocated storage could inadvertantly affect the memory that is mapped to the BSR register. Any such access will lead to unpredictable results.
- PE Version 5.1 requires LAPI Version 2.4.6 for AIX 5.3, and LAPI 3.1.2 for AIX 6.1. Earlier versions of LAPI are not supported.

Functional restrictions for PE for Linux 5.1:

Although many of the following functions, are currently available with Parallel Environment for AIX, they are not supported by Parallel Environment for Linux 5.1:

- Checkpoint and restart
- Lightweight core files
- Use of large memory pages
- PE does not support User Space jobs on IBM System x™ hardware.
- User Space jobs with Red Hat Enterprise Linux, when running on IBM Power Systems servers.
- The High Performance Computing Toolkit (HPC Toolkit) is not supported on IBM System x hardware.

Summary of changes

Changes for PE 5.1

This release of IBM Parallel Environment contains a number of functional enhancements.

The PE for AIX 5.1 enhancements are:

- For improved performance of on-node barrier synchronization, support for the IBM Power (POWER6) server barrier synchronization register (BSR) has been added. Note that you must be running 64-bit programs over the AIX 6.1 operating system, on IBM Power (POWER6) servers to utilize the BSR support. Note also that the MPI library will not use the BSR if checkpointing is enabled with the AIX environment variable **CHECKPOINT**. For more information, see *IBM Parallel Environment: MPI Programming Guide*.

- The default value of the **MP_PRIORITY_LOG** environment variable has changed from **yes** to **no**, so that the log file is produced only when it is needed.
- Beginning with PE 5.1, support for the PE Benchmarker has been removed. This includes the Performance Collection Tool (PCT), the Performance Visualization Tool (PVT), and the Unified Trace Environment (UTE) utilities **uteconvert**, **utemerge**, **utestats**, **traceTOslog2.so**, and **slogmerge**.
- To replace the performance analysis function of the PE Benchmarker, PE introduces the IBM High Performance Computing (HPC) Toolkit. The IBM HPC Toolkit is an integrated software environment that addresses the performance analysis, tuning, and debugging of sequential and parallel scientific applications. It consists of a collection of tools that optimize the application by monitoring its performance on the processor, memory, and network. The IBM HPC Toolkit is appropriate for users with varying degrees of parallel programming experience. For more information, see *IBM Parallel Environment: Operation and Use*.
- Beginning with PE 5.1, the pdbx debugger function has been removed. Instead, AIX users can now use the PDB debugger, previously available only with PE for Linux.
- With Version 5.1, PE introduces additional type checking for Fortran 90 codes. PE now includes a Fortran 90 module that provides type checking for MPI programs at compile time. This allows programmers to find and resolve errors at a much earlier stage.
- PE 5.1 enhances performance by providing a separate buffer for collective communication early arrival messages. Similar to **MP_BUFFER_MEM** for point-to-point communications, a new environment variable, **MP_CC_BUF_MEM**, allows users to control the amount of memory PE MPI allows for the buffering of early arrival message data for collective communications.

  Note: In PE 5.1, the early arrival buffer that is controlled by **MP_CC_BUF_MEM** is used by MPI_Bcast only. Early arrival messages in other collective communication operations continue to use the early arrival buffer for point-to-point communication that is controlled by **MP_BUFFER_MEM**.
- PE 5.1 is compliant with the revisions listed in the *Annex B Change-Log* of the MPI 2.1 standard.

The PE for Linux 5.1 enhancements are:

- The Parallel Operating Environment (POE) priority adjustment coscheduler, previously available only for AIX users, is now supported by PE for Linux.
- With Version 5.1, PE introduces additional type checking for Fortran 90 codes. PE now includes a Fortran 90 module that provides type checking for MPI programs at compile time. This allows programmers to find and resolve errors at a much earlier stage.
- PE 5.1 introduces the IBM High Performance Computing (HPC) Toolkit. The IBM HPC Toolkit is an integrated software environment that addresses the performance analysis, tuning, and debugging of sequential and parallel scientific applications. It consists of a collection of tools that optimize the application by monitoring its performance on the processor, memory, and network. The IBM HPC Toolkit is appropriate for users with varying degrees of parallel programming experience. For more information, see *IBM Parallel Environment: Operation and Use*.

- Beginning with PE 5.1, PDB is now available with both PE for Linux and PE for AIX.
- PE 5.1 enhances performance by providing a separate buffer for collective communication early arrival messages. Similar to **MP_BUFFER_MEM** for point-to-point communications, a new environment variable, **MP_CC_BUF_MEM**, allows users to control the amount of memory PE MPI allows for the buffering of early arrival message data for collective communications.

  Note: In PE 5.1, the early arrival buffer that is controlled by **MP_CC_BUF_MEM** is used by MPI_Bcast only. Early arrival messages in other collective communication operations continue to use the early arrival buffer for point-to-point communication that is controlled by **MP_BUFFER_MEM**.
- PE 5.1 is compliant with the revisions listed in the *Annex B Change-Log* of the MPI 2.1 standard.

Chapter 1. Introduction

The IBM Parallel Environment program product (PE) is an environment designed for developing and executing parallel Fortran, C, or C++ programs. PE consists of components and tools for developing, executing, debugging, profiling, and tuning parallel programs.

PE is a distributed memory message passing system. You can use PE to execute parallel programs on a variety of hardware, using the AIX or Linux operating system. For more information about the hardware and software that is supported with PE, refer to *IBM Parallel Environment: Installation*.

The processors of your system are called *processor nodes*. If you are using a Symmetric Multiprocessor (SMP) system, it is important to know that, although an SMP node has more than one processing unit, it is still considered, and referred to as, a *processor node*.

A parallel program executes as a number of individual, but related, *parallel tasks* on a number of your system's processor nodes. These *parallel tasks* taken together are sometimes referred to as a *parallel job*. The group of parallel tasks is called a *partition*. The processor nodes are connected on the same network, so the parallel tasks of your partition can communicate to exchange data or synchronize execution:

- Your system may have an optional high performance switch for communication. The switch increases the speed of communication between nodes. It supports a high volume of message passing with increased bandwidth and low latency.
- Your system may use the InfiniBand host channel adapter for improved I/O performance. PE only supports the InfiniBand host channel adapter on specific hardware. For more information, refer to *IBM Parallel Environment: Installation*.
- Your system administrator can divide its nodes into separate pools. A LoadLeveler system pool is a subset of processor nodes and is given an identifying pool name or number.

Note: The term high performance switch is used generically to refer to either the IBM High Performance Switch or the InfiniBand host channel adapters and switches, running on IBM clusters.

PE supports the two basic parallel programming models – SPMD and MPMD. In the *SPMD (Single Program Multiple Data) model*, the same program is running as each parallel task of your partition. The tasks, however, work on different sets of data. In the *MPMD (Multiple Program Multiple Data) model*, each task may be running a different program. A typical example of this is the master/worker MPMD program. In a master/worker program, one task – the master – coordinates the execution of all the others – the workers.

Note: While the remainder of this introduction describes each of the PE components and tools in relation to a specific phase of an application's life cycle, this does not imply that they are limited to one phase. They are ordered this way for descriptive purposes only; you will find many of the tools useful across an application's entire life cycle.

The application developer begins by creating a parallel program's source code. The application developer might create this program from scratch or could modify an existing serial program. In either case, the developer places calls to **Message Passing Interface (MPI)** or **Low-level Application Programming Interface (LAPI)** routines so that it can run as a number of parallel tasks. This is known as *parallelizing* the application. MPI provides message passing capabilities for the current version of PE Version 4.

Note: Throughout this information, when referring to anything not specific for MPI, the term *message passing* will be used. For example:

```
message passing program
message passing routine
message passing call
```

The message passing calls enable the parallel tasks of your partition to communicate data and coordinate their execution. The message passing routines, in turn, call the communication subsystem library routines which handle communication among the processor nodes. There are two separate implementations of the communication subsystem library – the Internet Protocol (IP) Communication Subsystem and the User Space (US) Communication Subsystem. While the message passing application interface remains the same, the communication subsystem libraries use different protocols for communication among processor nodes. The IP communication subsystem uses Internet Protocol, while the User Space communication subsystem is designed to exploit the high performance switch (for AIX) or the direct (kernel bypass) access to a high performance communication adapter (for Linux). The communication subsystem library implementations are dynamically loaded when you invoke the program. For more information on the message passing subroutine calls, refer to *IBM Parallel Environment: MPI Subroutine Reference* and *IBM Parallel Environment: Introduction*.

In addition to message passing communication, the Parallel Environment supports a separate communication protocol known as the **Low-level Application Programming Interface (LAPI)**. LAPI differs from MPI in that it is based on an *active message style* mechanism that provides a one-sided communications model. That is, the application at one process initiates an operation, and the completion of that operation does not require any other process to take an application-level complementary action.

LAPI is used as a common transport protocol for MPI, for both IP and User Space. For AIX, LAPI is part of Reliable Scalable Cluster Technology (RSCT), but is also shipped on the PE product CD (in the **rsct.lapi.rte** file set). For Linux, LAPI is shipped with PE for Linux in the following RPMs:

- 32-bit base IP
- 64-bit IP
- 32-bit US
- 64-bit US

Refer to the *IBM RSCT: LAPI Programming Guide* for more information.

After writing the parallel program, the application developer then begins a cycle of modification and testing. The application developer now compiles and runs his program from his **home node** using the **Parallel Operating Environment (POE)**. The home node can be any workstation on the LAN that has PE installed. POE is an execution environment designed to hide, or at least smooth, the differences between serial and parallel execution.

To assist with node allocation for job management, Tivoli Workload Scheduler (TWS) LoadLeveler (LoadLeveler) provides resource management function. AIX users can run parallel programs on a cluster of processor nodes running LoadLeveler or a clustered server that uses LoadLeveler. LoadLeveler not only provides node allocation for jobs using the User Space communication subsystem, but also provides management for other clustered nodes, or for nodes being used for jobs other than User Space. LoadLeveler can also be used for POE batch jobs. See *Tivoli Workload Scheduler LoadLeveler: Using and Administering* for more information on this job management system.

In general, with POE, you invoke a parallel program from your home node and run its parallel tasks on a number of **remote nodes**. As much as possible, the remote nodes should be managed to ensure that when they are running the tasks of your parallel program, none of them are being used for other activities. When you invoke a program on your home node, POE starts your **Partition Manager** which allocates the nodes of your partition and initializes the local environment. Depending on your hardware and configuration, the Partition Manager uses a **host list file**, LoadLeveler, or both a host list file and LoadLeveler to allocate nodes. A host list file contains an explicit list of node requests, while LoadLeveler can allocate nodes from one or more system pools implicitly based on their availability.

POE provides an option to enable you to specify whether your program will use MPI, LAPI, or both. Using this option, POE ensures that each API initializes properly and informs LoadLeveler which APIs are used so each node is set up completely.

For Single Program Multiple Data (SPMD) applications the Partition Manager executes the same program on all nodes. For Multiple Program Multiple Data (MPMD) applications, the Partition Manager prompts you for the name of the program to load as each task. The Partition Manager also connects standard I/O to each remote node so the parallel tasks can communicate with the home node. Although you are running tasks on remote nodes, POE allows you to continue using the standard UNIX®, AIX, or Linux execution techniques with which you are already familiar. For example, you can redirect input and output, pipe the output of programs, or use shell tools. POE includes:

- A number of **parallel compiler scripts**. These are shell scripts that call the C, C++, or Fortran compilers while also linking in an interface library to enable communication between your home node and the parallel tasks running on the remote nodes. You dynamically link in a communication subsystem implementation when you invoke the executable.
- A number of **POE Environment Variables** you can use to set up your execution environment. These are environment variables you can set to influence the operation of POE. These environment variables control such things as how processor nodes are allocated, what programming model you are using, and how standard I/O between the home node and the parallel tasks should be handled. Most of the POE environment variables also have associated command line flags that enable you to temporarily override the environment variable value when invoking POE and your parallel program.

For debugging parallel programs with PE, you can use PDB, in conjunction with DISH (Distributed Interactive Shell), POE, and GDB (for Linux) or dbx (for AIX). Note that there are two versions of GDB (the GNU Project debugger; *GDB* is for debugging 32-bit programs and *GDB64* is for 64-bit programs.

For more information about GNU's GDB, refer to the GNU Project Debugger Web site (**http://www.gnu.org/software/gdb/**).

After the parallel program has been debugged, you will want to tune the program for optimal performance. If you are running PE for Linux, you can use the Linux **gprof** command to do this. If you are using AIX, the parallel profiling capability of PE can help you analyze the program. The parallel profiling capability enables you to use the AIX Xprofiler graphical user interface, as well as the AIX **prof** and **gprof** commands on parallel programs.

Note: After the parallel program is tuned to your satisfaction, you might prefer to execute it using a job management system such as IBM LoadLeveler. If you do use a job management system, consult its documentation for information on its use.

PE Version 5 Release 1 migration information

If you are migrating from an earlier release of PE, you should be aware of some differences that you need to consider before installing and using PE Version 5 Release 1.

For information on PE for AIX migration, see "PE for AIX 5.1 migration information." For specific information on PE for Linux migration, see "PE for Linux 5.1 migration information" on page 8.

PE for AIX 5.1 migration information

To find out which release of PE you currently have installed, use the following command:

lslpp -ha *ppe.poe*

IBM Power (POWER6™) server support

PE Version 5 Release 1 provides support for IBM Power servers. The Power server installation may be either a standalone server, or a cluster of servers connected by an Ethernet LAN, running over IP.

Note that the IBM XL C/C++ Enterprise Edition for AIX compiler (formerly known as the VisualAge® C/C++ for AIX compiler), Version 8.0.0.12 (5724-M12) is required for Power server support.

AIX compatibility

PE Version 5 Release 1 commands and applications are compatible with AIX Version 5.3 and AIX Version 6.1, or later only, and not with earlier versions of AIX. AIX 6.1 is supported on standalone servers, running in IP mode only.

MPI library support

PE Version 5 Release 1 provides support for its threaded version of the MPI library only. An archive (libmpi.a) containing symbols resolving references made by non-threaded executables is also shipped to support binary compatibility. These merely map to the corresponding threaded library symbols.

Existing applications built as non-threaded applications will run as single threaded applications in the PE Version 5 Release 1 environment. Users and application developers should understand the implications of their programs running as threaded applications, as described in the *IBM Parallel Environment for AIX: MPI Programming Guide*.

LAPI support

LAPI is shipped on the PE CD, so you no longer need to obtain it from RSCT. There are three LAPI filesets shipped with PE Version 5 Release 1; **rsct.lapi.rte**, **rsct.lapi.samp**, and **rsct.lapi.bsr**. Refer to *IBM Parallel Environment: Installation* and the *IBM RSCT: LAPI Programming Guide* for more information on installing the LAPI filesets.

Additionally, MPI uses LAPI as a common transport protocol. If you are using the LAPI API to develop a message passing application, you may find useful information in the *IBM RSCT: LAPI Programming Guide*.

Binary compatibility

Binary compatibility is supported for existing applications that have been dynamically linked or created with the non-threaded compiler scripts from previous versions of POE. There is no support for statically bound executables.

Existing 32-bit applications that use striping may encounter memory usage conflicts and may need to be recompiled or use different run-time options in order to properly execute. See "Considerations for data striping, with PE" on page 86 for more information. 64-bit applications are not affected.

Obsolete POE environment variables and command line flags

The following environment variables and command line flags are obsolete:

- **MP_UTE** (used on PE compiler scripts only)
- **MP_BYTECOUNT** (used on PE compiler scripts only)

It is recommended that these variables and command line flags be removed from scripts and commands used to run POE applications.

Obsolete compiler script options

The following compiler script options should no longer be included in Makefiles:

- **-d7**
- **-lvtd** and **lvtd_r**

Note: If you are using Fortran and are making changes to your configuration files, it is important to ensure that those files do not contain references to obsolete flags, such as the ones listed above, or to stanzas that contain obsolete flags.

User Space applications with MP_EUIDEVICE/-euidevice (PE for AIX)

Existing User Space applications that set **MP_EUIDEVICE/-euidevice** to *sn_single* or *css0* on systems using multiple adapters and multiple networks will not benefit from the performance improvements provided by using the *sn_all* or *csss* value. In this case, you may want to change the **MP_EUIDEVICE/-euidevice** settings for such applications. Also note that **css1** can no longer be specified as a value for **MP_EUIDEVICE** or **-euidevice**. See "Step 3f: Set the MP_EUIDEVICE environment variable" on page 30 for more information.

Shared memory default changed

The use of shared memory for message passing between tasks running on the same node has been changed so that all invocations of POE will utilize shared memory as the default. For 64-bit applications, this includes using the shared memory enhanced collective communications algorithms. To run without using shared memory, change the value of the **MP_SHARED_MEMORY** environment variable or **-shared_memory** command line flag to **no**.

PSSP and the SP™ Switch are no longer supported
Beginning with PE 4.3, PSSP (Parallel System Support Programs) is no longer supported. Support for the SP Switch ended with the release of 4.2.

Task and memory affinity with LoadLeveler
Users that specify the **MP_TASK_AFFINITY** or **-task_affinity** POE options should be aware that with LoadLeveler 3.3.1 or later versions, LoadLeveler now handles scheduling affinity. As a result, memory and task affinity must be enabled in the LoadLeveler configuration file (using the **RSET_SUPPORT** keyword). In addition, **MP_TASK_AFFINITY** settings are ignored with batch jobs, and jobs requiring memory affinity must specify the appropriate LoadLeveler job control keywords to run with memory affinity. For more information, see "Managing task affinity on large SMP nodes" on page 57.

Some 32-bit applications that ran correctly before could fail with "out of memory" error
Some simple ways to work around this problem are suggested.

You might need to ensure that your system administrator has not established an entry in **/etc/security/limits** that might constrain memory allocations. It is suggested that the default stanza allow for unlimited resources, where practical.

Beginning with PE Version 5 Release 1, a new Early Arrival buffer for collective communications messages has been added. The new Early Arrival buffer is managed by the **MP_CC_BUF_MEM** environment variable. The default size of this new buffer is 4 MB, with an upper bound of 36 MB. A separate Early Arrival buffer for point-to-point messages still exists, and is managed by the **MP_BUFFER_MEM** environment variable (default value is now 64 MB). This is important to note because, by default, a 32-bit application might have a limited amount of memory available for this buffer. Applications may need to be compiled with **-bmaxdata** option to set aside more heap space.

The default size of the point-to-point Early Arrival buffer has been changed from 2.8 MB to 64 MB for 32-bit IP applications. The 2.8 MB used in previous releases can lead to performance problems when the job has more than a few tasks. A side effect of the new default could cause your application to fail due to insufficient memory.

By default, a 32-bit application can malloc approximately 200 MB before malloc fails. In previous releases, an IP application needed to allocate enough memory for the application itself, plus the 64 MB that was required for the point-to-point Early Arrival buffer. If the total amount of required memory was less than about 200 MB, the application ran correctly. However, now that a second Early Arrival buffer requires at least 4 MB of memory, IP applications that previously ran correctly might now fail due to insufficient memory. Also note that LAPI allocates 32 MB for **sn_single** and 64 MB for **sn_all,** so if you use **sn_all**, there is an additional 32 MB, which also counts against the 200 MB limit. In either of these circumstances, you may receive an out of memory error. In that case, you can recompile your application with the **-bmaxdata** option to set aside additional heap space, or use the **MP_BUFFER_MEM** environment variable (or **-buffer_mem** command line flag) to specify a size for the Early Arrival buffer that is smaller than the default of 64 MB. For more information about controlling the size of the point-to-point Early Arrival buffer, see "Using MP_BUFFER_MEM" on page 67. For more information about controlling the size of the new Early Arrival buffer for collective communication operations, see "MP_CC_BUF_MEM details" on page 240.

PE Benchmarker function has been removed
Beginning with PE Version 5 Release 1, support for the PE Benchmarker has been removed. This includes the Performance Collection Tool (PCT), the Performance Visualization Tool (PVT), and the Unified Trace Environment (UTE) utilities **uteconvert**, **utemerge**, **utestats**, **traceTOslog2.so**, and **slogmerge**.

PE debugging support
Beginning with PE Version 5 Release 1, the pdbx debugger function has been removed. Instead, AIX users can now use the PDB debugger, previously available only with PE for Linux.

Fortran 90 module support for type-checking at compile time
With Version 5 Release 1, PE introduces support for Fortran 90 *modules*. PE now includes a Fortran 90 module that provides type checking for MPI programs at compile time. This allows programmers to find and resolve errors at a much earlier stage.

In order to take advantage of the new compile time module, all Fortran 90 MPI programs must be:

- Modified to contain the following line:
  ```
  USE MPI
  ```
- Recompiled with XLF Version 12.1 or later. Note that if you have an existing Fortran MPI program that is running without errors, there is no need to recompile it. Since the program is already running correctly, compiling it with the Fortran 90 module would provide no benefit.

Note that Fortran 90 modules can only be used with the IBM Power Systems servers supported by PE.

POE LIBPATH default setting changed
POE will now use the value of the **MP_EUILIBPATH** environment variable, if it is specified, to automatically set the value of the **LIBPATH** environment variable. POE will no longer include the value of the **MP_EUILIB** setting in the **LIBPATH** value

MP_PRIORITY_LOG default setting changed
The default value of the **MP_PRIORITY_LOG** environment variable has been changed from **yes** to **no**. As a result, when the PE priority adjustment function is used, a priority adjustment coscheduler diagnostic log file is not created by default. You must now explicitly set **MP_PRIORITY_LOG** or **-priority_log** to **yes** in order for a diagnostic log to be produced. See "POE priority adjustment coscheduler" on page 97 for additional details.

New MP_SHMCC_EXCLUDE_LIST environment variable
A new environment variable, **MP_SHMCC_EXCLUDE_LIST**, is available for specifying the collective communication routines in which the MPI level shared memory optimization should be disabled. See Chapter 7, "POE Environment variables and command line flags," on page 215 for more information about **MP_SHMCC_EXCLUDE_LIST**.

AIX and Linux interoperability

PE does not support interoperability between nodes running AIX and Linux versions of PE. Parallel jobs cannot be mixed between AIX and Linux PE nodes.

PE for Linux 5.1 migration information

To find out which release of PE you currently have installed, use the following command:

perpms -h -a *ppe.poe*

Linux compatibility and coexistence

Version 5 Release 1 for Linux commands and applications are compatible with the supported distributions and levels of Linux, but not with earlier versions of Linux. Refer to *IBM Parallel Environment: Installation* for more information about the hardware and software levels that PE for Linux supports.

All nodes in a parallel job must run the same versions of PE, LoadLeveler, and LAPI at the same maintenance levels. It is recommended that you install and use PE Version 5 Release 1, LoadLeveler 3.5, and LAPI 2.4.6 (if using AIX 5.3) or LAPI 3.1.2 (if using AIX 6.1) at their latest support levels to provide the latest function.

Note: Mixing different node architectures (such as POWER5™, System x, x86, and Intel®) or different Linux distributions in a parallel job is not supported.

LAPI support

LAPI is shipped directly with PE as a set of RPMs which you install using the standard installation procedure.

Additionally, MPI uses LAPI as a common transport protocol. If you are using the LAPI API to develop a message passing application, you might find useful information in the *IBM RSCT: LAPI Programming Guide*.

Binary compatibility

Binary compatibility is supported for existing applications that have been dynamically linked or created with the non-threaded compiler scripts from previous versions of POE. There is no support for statically bound executables.

Existing 32-bit applications that use striping might encounter memory usage conflicts and might need to be recompiled or use different run-time options in order to properly execute. See “Considerations for data striping, with PE” on page 86 for more information. 64-bit applications are not affected.

Some 32-bit applications that ran correctly before could fail with ″out of memory″ error

Some simple ways to work around this problem are suggested.

You might need to ensure that your system administrator has not established an entry in **/etc/security/limits** that might constrain memory allocations. It is suggested that the default stanza allow for unlimited resources, where practical.

Beginning with PE Version 5 Release 1, a new Early Arrival buffer for collective communications messages has been added. The new Early

Arrival buffer is managed by the **MP_CC_BUF_MEM** environment variable. The default size of this new buffer is 4 MB, with an upper bound of 36 MB. A separate Early Arrival buffer for point-to-point messages still exists, and is managed by the **MP_BUFFER_MEM** environment variable (default value is now 64 MB). This is important to note because, by default, a 32-bit application might have a limited amount of memory available for this buffer. Applications might need to be compiled with **-bmaxdata** option to set aside more heap space.

Note: If you are using PE for AIX, note that 32-bit programs compiled for use with POE are limited to eight (8) data segments. The **-bmaxdata** option cannot specify more than 0x80000000. The actual amount available might be less, depending on whether shared memory or user space striping is being used by MPI or LAPI. See "Considerations for data striping, with PE" on page 86 for more information.

The default size of the point-to-point Early Arrival buffer has been changed from 2.8 MB to 64 MB for 32-bit IP applications. The 2.8 MB used in previous releases can lead to performance problems when the job has more than a few tasks. A side effect of the new default could cause your application to fail due to insufficient memory.

By default, a 32-bit application can malloc approximately 200 MB before malloc fails. In previous releases, an IP application needed to allocate enough memory for the application itself, plus the 64 MB that was required for the point-to-point Early Arrival buffer. If the total amount of required memory was less than about 200 MB, the application ran correctly. However, now that a second Early Arrival buffer requires at least 4 MB of memory, IP applications that previously ran correctly might now fail due to insufficient memory.

Also note that LAPI allocates 32 MB for **sn_single** and 64 MB for **sn_all**, so if you use **sn_all**, there is an additional 32 MB, which also counts against the 200 MB limit.

In either of these circumstances, you might receive an out of memory error. In that case, you can recompile your application with the **-bmaxdata** option to set aside additional heap space, or use the **MP_BUFFER_MEM** environment variable (or **-buffer_mem** command line flag) to specify a size for the Early Arrival buffer that is smaller than the default of 64 MB.

For more information about controlling the size of the point-to-point Early Arrival buffer, see "Using MP_BUFFER_MEM" on page 67. For more information about controlling the size of the new Early Arrival buffer for collective communication operations, see "MP_CC_BUF_MEM details" on page 240.

Fortran 90 module support for finding errors at compile time

With Version 5 Release 1, PE introduces support for Fortran 90 *modules*. PE now includes a Fortran 90 module that provides type checking for MPI programs at compile time. This allows programmers to find and resolve errors at a much earlier stage.

In order to take advantage of the new module, all Fortran 90 MPI programs must be:

- Modified to contain the following line:

```
USE MPI
```

- Recompiled with XLF Version 12.1 or later. Note that if you have an existing Fortran MPI program that is running without errors, there is no need to recompile it. Since the program is already running correctly, compiling it with the Fortran 90 module would provide no benefit.

New MP_SHMCC_EXCLUDE_LIST environment variable

A new environment variable, **MP_SHMCC_EXCLUDE_LIST**, is available for specifying the collective communication routines in which the MPI level shared memory optimization should be disabled. See Chapter 7, "POE Environment variables and command line flags," on page 215 for more information about **MP_SHMCC_EXCLUDE_LIST**.

AIX and Linux interoperability

PE does not support interoperability between nodes running AIX and Linux versions of PE. Parallel jobs cannot be mixed between AIX and Linux PE nodes.

Chapter 2. Executing parallel programs

POE is a simple and friendly environment designed to ease the transition from serial to parallel application development and execution. POE lets you develop and run parallel programs using many of the same methods and mechanisms as you would for serial jobs.

POE allows you to continue to use the standard UNIX, AIX, or Linux application development and execution techniques with which you are already familiar. For example, you can redirect input and output, pipe the output of programs into **more** or **grep**, write shell scripts to invoke parallel programs, and use shell tools such as **history**. You do all these in just the same way you would for serial programs. So while the concepts and approach to writing parallel programs must necessarily be different, POE makes your working environment as familiar as possible.

You can compile and execute your parallel C, C++, or Fortran programs on the supported hardware, as described in *IBM Parallel Environment: Installation*.

Executing parallel programs using POE

The first step in the life cycle of an application is actually writing the program. These instructions assume that you have already written your parallel C, C++, or Fortran program and, instead, describe the next step; compiling and executing it.

For information on writing parallel programs, refer to *IBM Parallel Environment: MPI Subroutine Reference*, *IBM Parallel Environment: MPI Programming Guide*, *IBM Cluster Systems Management: Command and Technical Reference*, and the *IBM RSCT: LAPI Programming Guide*.

Note: If you are using POE for the first time, check that you have authorized access. See *IBM Parallel Environment: Installation* for information on setting up users.

In order to execute an MPI or LAPI parallel program, you need to:

1. Compile and link the program using shell scripts or make files which call the C, C++, or Fortran compilers while linking in the Partition Manager interface and message passing subroutines.
2. Copy your executable to the individual nodes in your partition if it is not accessible to the remote nodes.
3. Set up your execution environment. This includes setting the number of tasks, and determining the method of node allocation.
4. Load and execute the parallel program on the processor nodes of your partition. You can:
   - load a copy of the same executable on all nodes of your partition. This is the normal procedure for SPMD programs.
   - individually load the nodes of your partition with separate executables. This is the normal procedure for MPMD programs.
   - load and execute a series of SPMD or MPMD programs, in job step fashion, on all nodes of your partition.

Step 1: Compile the program

As with a serial application, you must compile a parallel C, C++, or Fortran program before you can run it. Instead of using the usual programming commands such as **cc** (for AIX or Linux), and **xlC**, **xlf**, **cc_r**, **xlC_r**, or **xlf_r** (for AIX), you use commands that not only compile your program, but also link in the Partition Manager and message passing interface libraries. When you later invoke the program, the subroutines in these libraries enable the home node Partition Manager to communicate with the parallel tasks, and the tasks with each other.

PE for AIX parallel programs can also utilize functions to checkpoint and later restart a program. For more information on checkpointing refer to "Checkpointing and restarting programs (PE for AIX only)" on page 54.

For each of the supported compilers – C, C++, Fortran, Fortran 90, Fortran 95 (PE for AIX only), and Fortran 2003 (PE for AIX only), POE provides separate commands to compile and link application programs with the parallel libraries, allowing the program to run in parallel. To compile a program for use with POE, you use the compilers below. These commands generate thread-aware code by linking in the threaded version of MPI, including the threaded POE utility library.

PE for AIX compilers:
- **mpcc_r** (C compiler)
- **mpCC_r** (C++ compiler)
- **mpxlf_r** (Fortran compiler)
- **mpxlf90_r** (Fortran 90 compiler)
- **mpxlf95_r** (Fortran 95 compiler)
- **mpxlf2003_r** (Fortran 2003 compiler)

PE for Linux compilers:
- **mpcc** (C compiler)
- **mpCC** (C++ compiler)
- **mpfort** (Fortran compiler)

Note: IBM XL Fortran 90 support on PE for Linux is limited to XLF Version 9.1.0-1 or later, on the IBM Power Systems hardware that is currently supported by PE, with the IBM C Set++ compiler suite installed.

The POE compiler scripts create dynamically bound executables, referencing the appropriate MPI, LAPI, and threaded libraries, some of which are dynamically loaded. As a result, it is not possible to create statically bound executables. Note that PE for AIX no longer supports the use of statically bound application programs.

In previous releases of PE for AIX, there were two versions of these commands, for non-threaded and threaded programs. Only the threaded version of MPI is supported. Legacy POE scripts, such as **mpcc**, **mpCC**, and **mpxlf**, are now symbolic links to **mpcc_r**, **mpCC_r**, and **mpxlf_r**.

The compiler commands are actually shell scripts that call the appropriate compiler. You can use any of the **cc_r**, **xlC_r**, or **xlf_r** flags on these commands. We suggest you allow the scripts to provide appropriate include paths for the PE MPI include files rather than provide them explicitly.

Table 4 and Table 5 show the command to enter to compile a program, depending on the language in which it is written and the operating system you are using. For more information on these commands, see Chapter 6, "Parallel Environment commands," on page 129.

Table 4. Compiling a PE for AIX program

To compile:	ENTER
a C program	**mpcc_r** *program.c* **-o** *program*
a C++ program (that does not use the IBM PE MPI class structure)	**mpCC_r** *program.C* **-o** *program*
a C++ program (that uses the IBM PE MPI class structure)	**mpCC_r -cpp -o** *program program.C*
a Fortran program	**mpxlf_r** *program.f* **-o** *program*
a Fortran 90 program	**mpxlf90_r** *program.f* **-o** *program*
a Fortran 95 program	**mpxlf95_r** *program.f* **-o** *program*
a Fortran 2003 program	**mpxlf2003_r** *program.f* **-o** *program*

Table 5. Compiling a PE for Linux program

To compile:	ENTER
a C program	**mpcc** *program.c* **-o** *program*
a C++ program (that does not use the IBM PE MPI class structure)	**mpCC** *program.C* **-o** *program*
a C++ program (that uses the IBM PE MPI class structure)	**mpCC -cpp -o** *program program.C*
a Fortran program	**mpfort** *program.f* **-o** *program*

PE Fortran 90 module for type-checking at compile time

The IBM MPI implementation includes hundreds of functions, several of which have arguments that do not conform to any particular data type. In the past, because there was no type-checking mechanism available at compile time, these arguments could potentially be used with any of the available Fortran data types. Without type-checking, an MPI function that did not call the correct number of arguments compiled with no errors, but often crashed later due to a segmentation fault.

However, starting with Fortran 90, the Fortran standard introduced support for *modules*. A Fortran 90 module is a type of program unit that can provide instructions for declaring and defining interfaces. By using a module to define the interface, a programmer can enforce the data types that a function can accept. PE now provides a Fortran 90 module for type-checking at compile time (**mpi.mod**) that can be called by a Fortran 90 MPI program. When using the module, any unsolicited MPI function call that does not conform to the interface definition generates an error when the program is compiled.

To use the PE Fortran 90 type-checking module, do the following.

For new programs:
- Include the `USE MPI` statement to the application source code.
- Compile the program with XLF Version 12.1 or later.

For existing programs:

- Modify the application source code to include the `USE MPI` statement.
- Recompile the program with XL Fortran compiler (XLF) Version 12.1 or later. Note that if you have an existing Fortran MPI program that is running without errors, there is no need to recompile it. Since the program is already running correctly, compiling it with the Fortran 90 module would provide no benefit.

The PE Fortran 90 type-checking module is only supported by PE on IBM Power Systems servers running the supported levels of AIX and Linux. XL Fortran 90 Version 12.1 or later must be installed.

Notes on compiling:

1. Be sure to specify the **-g** flag when compiling a program for use with the debugger. The **-g** flag is a standard compiler flag that produces an object file with symbol table references. These symbol table references are needed by the debugger.

   For more information about using the **-g** option, AIX users should refer to its use on the **cc** command as described in *IBM AIX Commands Reference*. Linux users should refer to the documentation for the compiler they are using.
2. If you are using PE for AIX, note that 32-bit programs compiled for use with POE are limited to eight (8) data segments. The **-bmaxdata** option cannot specify more than 0x80000000. The actual amount available might be less, depending on whether shared memory or user space striping is being used by MPI and/or LAPI. See "Considerations for data striping, with PE" on page 86 for more information.
3. The POE compiler scripts will evaluate a dollar sign ($) in a file name as if it were a shell variable, which might not produce the desired result in resolving the file name to be compiled. If your program file names contain the dollar sign, you will need to prevent the compiler scripts from evaluating it as a shell variable.

   For example, if your file name is *$foo.f*, you need to invoke the compiler script as shown below.

   For PE for AIX:
   ```
   mpxlf_r "\\\$foo.f"
   ```
   ```
   mpxlf_r "*foo.f"
   ```
   For PE for Linux:
   ```
   mpfort "\\\$foo.f"
   ```
   or
   ```
   mpfort "*foo.f"
   ```
4. With PE for AIX, POE compile scripts utilize the **-binitfini** binder option. As a result, POE programs have a priority default of zero. If other user applications are using the **initfini** binder option, they should only specify a priority in the range of 1 to 2,147,483,647.
5. Beginning with PE 5.1, a new Early Arrival buffer for collective communications messages has been added. The default size of this new buffer is 4 MB, but the **MP_CC_BUF_MEM** environment variable allows users to specify a maximum value of 128 MB. A separate Early Arrival buffer for point-to-point messages still exists, and is managed by the **MP_BUFFER_MEM** environment variable (default value is 64 MB). This is important to note because, by default, a 32-bit application might have a limited amount of memory available for this buffer. Applications might need to be compiled with

-bmaxdata option to set aside more heap space. For more information, see "PE Version 5 Release 1 migration information" on page 4.

Step 2: Copy files to individual nodes

Note: You only need to perform this step if your executable, your data files, and (if you plan to use a debugger) your source code files are not in a commonly accessed, *shared*, or parallel file system.

If the program you are running is in a shared file system, the Partition Manager loads a copy of your executable in each processor node in your partition when you invoke a program. If your executable is in a private file system, however, you must copy it to the nodes in your partition. If you plan to use a debugger, you must copy your source files to all nodes as well.

You can copy your executable to each node with the **mcp** command. **mcp** uses the message passing facilities of the Parallel Environment to copy a file from a file system on the home node to a remote node file system. For example, assume that your executable program is on a mounted file system (*/u/edgar/somedir/myexecutable*), and you want to make a private copy in */tmp* on each node in *host.list*.

ENTER

mcp */u/edgar/somedir/myexecutable /tmp/myexecutable* **-procs** *n*

For more information on the **mcp** command, refer to "mcp" on page 143.

Note: If you load your executable from a mounted file system, you may experience an initial delay while the program is being initialized on all nodes. You may experience this delay even after the program begins executing, because individual pages of the program are brought in on demand. This is particularly apparent during initialization of a parallel program; since individual nodes are synchronized, there are simultaneous demands on the network file transfer system. You can minimize this delay by copying the executable to a local file system on each node, using the **mcp** message passing file copy program.

Step 3: Set up the execution environment

This step contains the following sections:
- "Step 3a: Set the MP_PROCS environment variable" on page 21
- "Step 3b: Create a host list file" on page 22
- "Step 3c: Set the MP_HOSTFILE environment variable" on page 27
- "Step 3d: Set the MP_RESD environment variable" on page 28
- "Step 3e: Set the MP_EUILIB environment variable" on page 29
- "Step 3f: Set the MP_EUIDEVICE environment variable" on page 30
- "Step 3g: Set the MP_DEVTYPE environment variable" on page 32
- "Step 3h: Set the MP_MSG_API environment variable" on page 32
- "Step 3i: Set the MP_RMPOOL environment variable" on page 33

Before invoking your program, you need to set up your execution environment. The POE environment variables are summarized in Chapter 7, "POE Environment variables and command line flags," on page 215. Any of these environment variables can be set at this time to later influence the execution of parallel programs.

This step covers the environment variables that are most important to successfully invoke a parallel program. When you invoke a parallel program, your home node Partition Manager checks these environment variables to determine:

- the number of tasks in your program as specified by the **MP_PROCS** environment variable.
- how to allocate processor nodes for these tasks. There are two basic methods of node allocation – specific and nonspecific.

  For *specific node allocation*, the Partition Manager reads an explicit list of nodes contained in a *host list* file you create. If you do not have LoadLeveler, or if you are using nodes that are not part of the LoadLeveler cluster, you must use this method of node allocation.

  For *nonspecific node allocation*, you give the Partition Manager the name or number of a LoadLeveler pool. A pool name or number may also be provided in a host list file. The Partition Manager then connects to LoadLeveler, which allocates nodes from the specified pool(s) for you. For more information on LoadLeveler and LoadLeveler pools, refer to the scenarios for allocating nodes with LoadLeveler in "Considerations for using the High Performance Switch interconnect" on page 78.

The architectural limits for Parallel Environment Version 5.1 are as follows.

- 16 K tasks
- 1500 nodes
- 128 tasks per node for shared memory communications.

Supported limits for Parallel Environment Version 5.1 for a particular platform can be obtained by going to the appropriate Sales Manual (**http://www.ibm.com/common/ssi/index.wss**) and entering the desired product number). The architectural limits represent a ceiling. Support for larger task counts on any particular platform may be available by special bid. Contact your IBM representative.

Note: The IBM High Performance Switch, when running over User Space, is limited to 64 tasks per adapter (128 tasks per node with two adapters per network).

There are six separate environment variables that, collectively, determine how nodes are allocated by the Partition Manager. The following description of these environment variables assumes that you are **not** submitting a job using a LoadLeveler job command file as described in "Submitting an interactive POE job using a TWS LoadLeveler command file" on page 76. If you do intend to use a LoadLeveler job command file, be aware that, in order to avoid conflicting allocation specifications made via POE environment variables/command line flags, LoadLeveler job command file statements, and POE host list file entries, certain settings will be ignored or will cause errors. The following information, therefore, assumes that you are not using a LoadLeveler job command file. Also keep in mind that, while the following environment variables are the only ones you must set to allocate nodes, there are many other environment variables you can set. These are summarized in Chapter 7, "POE Environment variables and command line flags," on page 215, and control such things as standard I/O handling and message passing information. The environment variables for node allocation are:

MP_HOSTFILE
which specifies the name of a host list file to use for node allocation. If set to an empty string (" ") or to the word "*NULL*", this environment variable specifies that no host list file should be used. If **MP_HOSTFILE** is not set, POE looks for a file *host.list* in the current directory. You need to create a host list file if you want specific node allocation.

MP_RESD
which specifies whether or not the Partition Manager should connect to LoadLeveler to allocate nodes.

Note: When running POE from a workstation that is external to the LoadLeveler cluster, the **LoadL.so** file set (AIX) or **LoadL-so-***Linux_identifer***.rpm** (Linux) must be installed on the external node (see *Tivoli Workload Scheduler LoadLeveler: Using and Administering* and *IBM Parallel Environment: Installation* for more information).

MP_EUILIB
which specifies the communication subsystem implementation to use – either the IP communication subsystem implementation or the User Space (US) communication subsystem implementation. The IP communication subsystem uses Internet Protocol for communication among processor nodes, while the User Space communication subsystem lets you drive a clustered server's high-speed interconnect switch (AIX) or a high performance communication adapter (Linux) directly from your parallel tasks, without going through the kernel or operating system. For User Space communication on a clustered server system, you must have the high-speed interconnect switch feature.

MP_EUIDEVICE
which specifies the adapter set or the adapter device name or network type, configured in LoadLeveler, that you want to use for communication among processor nodes. The Partition Manager checks this if you are using the communication subsystem implementation with LoadLeveler. If **MP_RESD**=*no*, the value of **MP_EUIDEVICE** is ignored. For User Space, the value **css0** (AIX only) or **sn_single** (AIX and Linux) specifies that windows are requested on one common network. The value **csss** (AIX) or **sn_all** (AIX or Linux) specifies that windows are requested from each network in the system. The number of windows being requested depends on the value of the **MP_INSTANCES** environment variable (the default is one). In the case of **csss** and **sn_all**, the number of windows being requested also depends on the number of networks in the system.

MP_DEVTYPE
which specifies the device type class; InfiniBand. To specify InfiniBand support, set **MP_DEVTYPE** to *ib*. There is no default value.

Note: For AIX, InfiniBand is only supported on the IBM BladeCenter® JS21 Express server and the IBM POWER5 server. For Linux, InfiniBand is not supported on the IBM System x servers or the IBM BladeCenter JS20 Express or IBM BladeCenter JS21 Express servers.

MP_RMPOOL
which specifies the name or number of a LoadLeveler pool. The Partition Manager only checks this if you are using LoadLeveler without a host list file. You can use the **llstatus** command to return information about

LoadLeveler pools. To use **llstatus** on a workstation that is external to the LoadLeveler system, the following must be installed on the external node:

- For AIX, the **LoadL.so** fileset
- For Linux, **LoadL-so-***linux_identifier***.rpm**

For more information, see *Tivoli Workload Scheduler LoadLeveler: Using and Administering* and *IBM Parallel Environment: Installation*.

The remainder of this step consists of sub-steps describing how to set each of these environment variables, and how to create a host list file. Depending on the hardware and message passing library you are using, and the method of node allocation you want, some of the sub-steps that follow may not apply to you. For this reason, pay close attention to the task variant tables at the beginning of many of the sub-steps. They will tell you whether or not you need to perform the sub-step.

For further clarification, the following tables summarize the procedure for determining how nodes are allocated. The tables describe the possible methods of node allocation available to you, to what each environment variable must be set, and whether or not you need to create a host list file.

As already stated, these instructions assume that you are **not** using a LoadLeveler job command file and, therefore, the **MP_LLFILE** environment variable (or its associated command line flag **-llfile**) is **not** set. To allocate nodes using a LoadLeveler job command file, refer to "Submitting an interactive POE job using a TWS LoadLeveler command file" on page 76 or the manual *Tivoli Workload Scheduler LoadLeveler: Using and Administering*.

To make the procedure of setting up the execution environment easier and less prone to error, you may eventually wish to create a shell script which automates some of the environment variable settings. To allocate the nodes of a clustered server that uses LoadLeveler, see Table 6 and Table 7 on page 19. If you are using a network cluster (or, for AIX users, a mixed system) and want to allocate some nodes that are not part of the LoadLeveler cluster, see Table 8 on page 19.

If you want to use the User Space communication subsystem library for communication among parallel tasks, and...

Table 6. Execution setup summary for User Space (for a clustered server with LoadLeveler)

You want specific node allocation:	**You want nonspecific node allocation from a single LoadLeveler pool:**
A host list file is required.	A host list file is not required. If used, however, all entries must specify the same LoadLeveler pool.
MP_HOSTFILE should be set to the name of your host list file. If not set, the host list file is assumed to be *host.list* in the current directory.	A host list file is not required. If none is used, **MP_HOSTFILE** should be set to an empty string ("") or the word "NULL".
MP_RESD should be set to *yes*. If set to an empty string (""), or if not set, the Partition Manager assumes the value of **MP_RESD** is *yes*.	**MP_RESD** should be set to *yes*. If set to an empty string (""), or if not set, the Partition Manager assumes the value of **MP_RESD** is *yes*.
MP_EUILIB should be set to *us*. The values of **MP_EUILIB** are case-sensitive.	**MP_EUILIB** should be set to *us*. The values of **MP_EUILIB** are case-sensitive.
MP_EUIDEVICE should be set to: • **For AIX users:** *sn_all*, *sn_single*, *csss*, *css0* • **For Linux users:** *sn_all*, *sn_single*	**MP_EUIDEVICE** should be set to: • **For AIX users:** *sn_all*, *sn_single*, *csss*, *css0* • **For Linux users:** *sn_all*, *sn_single*

Table 6. Execution setup summary for User Space (for a clustered server with LoadLeveler) (continued)

You want specific node allocation:	You want nonspecific node allocation from a single LoadLeveler pool:
MP_RMPOOL is ignored because you are using a host list file.	if you are not using a host list file, **MP_RMPOOL** should be set to the name or number of a LoadLeveler pool. If you are using a host list file, **MP_RMPOOL** is ignored; you must specify the pool in the host list file.

If you want to use the IP communication subsystem library for communication among parallel tasks, and...

Table 7. Execution setup summary for IP (for a clustered server with LoadLeveler)

You want specific node allocation:	You want nonspecific node allocation from a single LoadLeveler pool:
A **host list file** is required.	A **host list file** is not required. If used, however, all entries must specify the same LoadLeveler pool.
MP_HOSTFILE Should be set to the name of your host list file. If not set, the host list file is assumed to be *host.list* in the current directory.	No host list file is required. If none is used, **MP_HOSTFILE** should be set to an empty string ("") or the word "NULL".
MP_RESD should be set to *yes*. If set to an empty string (""), or if not set, the Partition Manager assumes the value of **MP_RESD** is *no*.	**MP_RESD** should be set to *yes*. If set to an empty string (""), or if not set, the Partition Manager assumes the value of **MP_RESD** is *yes*.
MP_EUILIB should be set to *ip*. The values of **MP_EUILIB** are case-sensitive.	**MP_EUILIB** should be set to *ip*. The values of **MP_EUILIB** are case-sensitive.
MP_EUIDEVICE should specify the adapter type.	**MP_EUIDEVICE** should specify the adapter type.
MP_RMPOOL is ignored because you are using a host list file.	if you are not using a host list file, **MP_RMPOOL** should be set to the name or number of a LoadLeveler pool. If you are using a host list file, **MP_RMPOOL** is ignored; you must specify the pool in the host list file.

Note: This preceding table assumes that the **MP_LLFILE** environment variable is not set, and the **-llfile** flag is not used. If the **MP_LLFILE** environment variable (or its associated command line flag) is used, indicating that a LoadLeveler job command file should participate in node allocation, be aware that some of the environment variables shown in this table will be ignored. The reason they will be ignored is to avoid conflicting allocation specifications made via POE environment variables/command line flags, POE host list file entries, and LoadLeveler job command file statements. For more information on the POE environment variables that will be ignored when a LoadLeveler job command file is used, refer to "Submitting an interactive POE job using a TWS LoadLeveler command file" on page 76.

Table 8 summarizes the execution environment setup for an IBM Power Systems cluster or a mixed system, whose additional nodes are not part of the LoadLeveler cluster. In this scenario, a host list file must be used.

Table 8. Execution environment setup summary (for an IBM Power Systems network cluster or a mixed system, whose additional nodes are not part of the LoadLeveler cluster)

This environment variable...	is set as follows
MP_HOSTFILE	should be set to the name of a host list file. If not defined, the host list file is assumed to be *host.list* in the current directory.

Table 8. Execution environment setup summary (for an IBM Power Systems network cluster or a mixed system, whose additional nodes are not part of the LoadLeveler cluster) (continued)

This environment variable...	**is set as follows**
MP_RESD	should be set to *no*.
MP_EUILIB	should be set to *ip*.
MP_RMPOOL	is not used because you are using a host list file.

Table 9 shows how nodes are allocated depending on the value of the environment variables discussed in this step. It is provided here for additional illustration. Refer to it in situations when the environment variables are set in patterns other than those suggested in Table 6 on page 18, Table 7 on page 19, and Table 8 on page 19. When reading Table 9, be aware that, if a LoadLeveler job command file is specified (using the **MP_LLFILE** environment variable or the **-llfile** flag), the value of **MP_RESD** will be *yes*.

Table 9. Node allocation summary

If			**Then**		
The value of MP_EUILIB is:	**The value of MP_RESD is:**	**Your Host List file contains a list of:**	**The allocation mode will be:**	**The communication subsystem library implementation used will be:**	**The message passing address used will be:**
ip	-	nodes	Node_List	IP	Nodes
		pools	LL_List	IP	MP_EUIDEVICE
		NULL	LL	IP	MP_EUIDEVICE
	yes	nodes	LL_List	IP	MP_EUIDEVICE
		pools	LL_List	IP	MP_EUIDEVICE
		NULL	LL	IP	MP_EUIDEVICE
	no	nodes	Node_List	IP	Nodes
		pools	Error	-	-
		NULL	Error	-	-
us	-	nodes	LL_List	US	N/A
		pools	LL_List	US	N/A
		NULL	LL	US	N/A
	yes	nodes	LL_List	US	N/A
		pools	LL_List	US	N/A
		NULL	LL	US	N/A
	no	nodes	Node_List IP Nodes	-	-
		pools	Error	-	-
		NULL	Error	-	-

Table 9. Node allocation summary (continued)

If			Then		
-	-	nodes	Node_List	IP	Nodes
		pools	LL_List	IP	MP_EUIDEVICE
		NULL	LL	IP	MP_EUIDEVICE
	yes	nodes	LL_List	IP	MP_EUIDEVICE
		pools	LL_List	IP	MP_EUIDEVICE
		NULL	LL	IP	MP_EUIDEVICE
	no	nodes	Node_List	IP	Nodes
		pools	Error	-	-
		NULL	Error	-	-

Table notes:

Node_List
means that the host list file is used to create the partition.

LL_List
means that the host list file is used to create the partition, but the nodes are requested from LoadLeveler.

LL means that the partition is created by requesting nodes in **MP_RMPOOL** from LoadLeveler.

Nodes indicates that the external IP address of the processor node is used for communication.

MP_EUIDEVICE
indicates that the IP adapter address indicated by **MP_EUIDEVICE** is used for communication.

Step 3a: Set the MP_PROCS environment variable

Before you execute a program, you need to set the size of the partition. To do this, use the **MP_PROCS** environment variable or its associated command line flag **-procs**, as shown in Table 10.

For example, say you want to specify the number of task processes as 6. You could:

Table 10. Example of setting the MP_PROCS environment variable or -procs command line flag

Set the MP_PROCS environment variable:	Use the -procs flag when invoking the program:
ENTER **export MP_PROCS=6**	**ENTER** **poe** *program* **-procs 6**

If you do not set **MP_PROCS**, the default number of task processes is 1 unless you have set the **MP_RMPOOL** environment variable (or **-rmpool** command line flag) for nonspecific node allocation from a single LoadLeveler pool, and have set both the **MP_NODES** and **MP_TASKS_PER_NODE** environment variables (or their associated command line flags) to further specify how LoadLeveler should allocate nodes within the pool. In such cases, if **MP_PROCS** is not set, the parallel job will consist of **MP_TASKS_PER_NODE** multiplied by **MP_NODES** tasks. See "Step 3i: Set the MP_RMPOOL environment variable" on page 33 for more details.

Step 3b: Create a host list file

A host list file specifies the processor nodes on which the individual tasks of your program should run. When you invoke a parallel program, your Partition Manager checks to see if you have specified a host list file. If you have, it reads the file to allocate processor nodes.

You need to create a host list file if you are **not** using LoadLeveler to allocate nodes.

The procedure for creating a host list file differs, depending on whether you are using LoadLeveler. If you are **not** using LoadLeveler, see "Creating a host list file to allocate nodes of a cluster without LoadLeveler." If you are using LoadLeveler, see "Creating a host list file to allocate nodes with LoadLeveler" on page 23.

Creating a host list file to allocate nodes of a cluster without LoadLeveler:

If you are not using LoadLeveler to allocate nodes, you must create a host list file, which simply lists a series of host names – one per line. These must be the names of remote nodes accessible from the home node. If you plan to use the IP communication subsystem with the InfiniBand interconnect, the InfiniBand adapter host names should be specified as host list file entries. Each line specifies where one task is to be run so when SMP nodes are to run multiple tasks, the same node name can appear more than once. Lines beginning with an exclamation point (!) or asterisk (*) are comments. The Partition Manager ignores blank lines and comments. The host list file can list more names than are required by the number of program tasks. The additional names are ignored.

To understand how the Partition Manager uses a host list file to determine the nodes on which your program should run, consider the following example host list file:

```
! Host list file for allocating 6 tasks

* An asterisk may also be used to indicate a comment

host1_name
host2_name
host3_name
host4_name
host5_name
host6_name
```

The Partition Manager ignores the first two lines because they are comments, and the third line because it is blank. It then allocates *host1_name* to run task 0, *host2_name* to run task 1, *host3_name* to run task 2, and so on. If any of the processor nodes listed in the host list file are unavailable when you invoke your program, the Partition Manager returns a message stating this and does not run your program.

You can also have multiple tasks of a program share the same node by simply listing the same node multiple times in your host list file. For example, say your host list file contains the following:

```
host1_name
host2_name
```

```
host3_name
host1_name
host2_name
host3_name
```

Tasks 0 and 3 will run on *host1_name*, tasks 1 and 4 will run on *host2_name*, and tasks 2 and 5 will run on *host3_name*.

Creating a host list file to allocate nodes with LoadLeveler:

If you are using LoadLeveler to allocate nodes, you can use a host list file for either:
- nonspecific node allocation from one system pool only.
- specific node allocation.

In either case, the host list file can contain a number of records – one per line. For specific node allocation, each record indicates a processor node. For nonspecific node allocation you can have one system pool only. Your host list file cannot contain a mixture of node and pool requests, so you must use one method or the other. The host list file can contain more records than required by the number of program tasks. The additional records are ignored.

For specific node allocation:

Each record is either a host name or IP adapter address of a specific processor node of the system. If you are using a mixed system and want to allocate nodes that are not part of the LoadLeveler cluster, you must request them by host name. Lines beginning with an exclamation point (!) or asterisk (*) are comments. The Partition Manager ignores blank lines and comments.

To understand how the Partition Manager uses a host list file to determine the system nodes on which your program should run, consider the following representation of a host list file.

```
! Host list file for allocating 6 tasks

host1_name
host2_name
host3_name
9.117.8.53
9.117.8.53
9.117.8.53
```

The Partition Manager ignores the first line because it is a comment, and the second because it is blank. It then allocates *host1_name* to run task 0, *host2_name* to run task 1, *host3_name* to run task 2, and so on. The last three nodes are requested by adapter IP address using dot decimal notation.

Note: If any of the processor nodes listed in the host list file are unavailable when you invoke your program, the Partition Manager returns a message stating this and does not run your program.

For nonspecific node allocation from a LoadLeveler pool:

After installation of a LoadLeveler cluster, your system administrator divides its processor nodes into a number of pools. With LoadLeveler, each pool has an identifying pool name or number. Using LoadLeveler for nonspecific node allocation, you need to supply the appropriate pool name or number. When specifying pools in a host list file, each entry must be for the same pool.

If you require information about LoadLeveler pools, use the command **llstatus**. To use **llstatus** on a workstation that is external to the LoadLeveler cluster, the following must be installed on the external node:

- For AIX users: **LoadL.so** fileset
- For Linux users: **LoadL-so-*linux_identifer*.rpm**

See *Tivoli Workload Scheduler LoadLeveler: Using and Administering* for more information.

ENTER
llstatus -l (lower case L)

LoadLeveler lists status information including the pools in the LoadLeveler cluster.

For more information on the **llstatus** command and LoadLeveler pools, see *Tivoli Workload Scheduler LoadLeveler: Using and Administering*.

When specifying LoadLeveler pools in a host list file, each entry must refer to the same pool (by name or number), and should be preceded by an at symbol (@). Lines beginning with an exclamation point (!) and asterisk (*) are comments. The Partition Manager ignores blank lines and comments.

To understand how the Partition Manager uses a host list file for nonspecific node allocation, consider the following example host list file:

```
! Host list file for allocating 3 tasks with LoadLeveler

@6
@6
@6
```

The Partition Manager ignores the first line because it is a comment, and the second line because it is blank. The at (@) symbols tell the Partition Manager that these are pool requests. It connects to LoadLeveler to request three nodes from pool 6.

Note: If there are insufficient nodes available in a requested pool when you invoke your program, the Partition Manager returns a message stating this, and does not run your program.

Specifying how a node's resources are used:

When requesting nodes using LoadLeveler specific node allocation, you can optionally request how each node's resources – its adapters and CPU – should be used. You can specify:

- Whether the node's adapter is to be *dedicated* or *shared*.

  If the node's adapter is to be dedicated, and if using:

  – A single adapter, then only a single program task can use it for the same protocol.
  – Striping or multiple adapters, then any window that is allocated on an adapter will prevent other tasks from using windows on the same adapter.

  If the node's adapter is to be *shared*, a number of tasks of different jobs on that node can use it. (see Table 11 on page 25).

- Whether the node's CPU usage should be *unique* or *multiple*. If *unique*, only your program's tasks can use the CPU. If *multiple*, your program may share the node with other users.

If dedicated, using a single adapter, only a single program task can use it for the same protocol. If dedicated, using multiple adapters, or if using striping, any window that is allocated on an adapter will prevent other tasks from using windows on the same adapter.

When using LoadLeveler for nonspecific node allocation, any usage specification in the host list file will be ignored. Instead, you can request how nodes are used with the **MP_CPU_USE** and/or **MP_ADAPTER_USE** environment variables (or their associated command line options) or you can specify this information in a LoadLeveler Job Command File.

Using the environment variables **MP_ADAPTER_USE** and **MP_CPU_USE**, or the associated command line options (**-adapter_use** and **-cpu_use**) to make either or both of these specifications will affect the resource usage for each node allocated from the pool specified using **MP_RMPOOL** or **-rmpool**. For example, if you wanted nodes from pool 5, and you wanted your program to have exclusive use of both the adapter and CPU, the following command line could be used:

poe [*program*] *-rmpool 5 -adapter_use*[*dedicated*]

-cpu_use[*unique*] [*more_poe_options*]

Associated environment variables (**MP_RMPOOL**, **MP_ADAPTER_USE**, **MP_CPU_USE**) could also be used to specify any or all of the options in this example.

Note: You can also use a LoadLeveler job command file to specify how a node's resources are used. If you use a LoadLeveler job command file, the **MP_RMPOOL**, **MP_ADAPTER_USE**, and **MP_CPU_USE** environment variables will be validated but ignored. For more information about LoadLeveler job command files, see *Tivoli Workload Scheduler LoadLeveler: Using and Administering*.

Table 11 and Table 12 on page 26 illustrate how node resources are used. Table 11 shows the default settings for adapter and CPU use, while Table 12 on page 26 outlines how the two separate specifications determine how the allocated node's resources are used.

Table 11. Adapter/CPU default settings

Dedicated/shared adapter	Adapter	CPU
If host list file contains nonspecific pool requests:	Dedicated	Unique
If host list file requests specific nodes:	Shared (for User Space jobs, adapter is dedicated)	Multiple
If host list file is not used:	Dedicated (for IP jobs, adapter is shared)	Unique (for IP jobs, CPU is multiple)

Table 12. Adapter/CPU use under LoadLeveler

Adapter use	If the Node's CPU is "Unique":	If the Node's CPU is "Multiple":
If the adapter use is "Dedicated":	Intended for production runs of high performance applications. Only the tasks of that parallel job use the adapter and CPU.	The adapter you specified with **MP_EUIDEVICE** is dedicated to the tasks of your parallel job. However, you and other users still have access to the CPU through another adapter. Also, if you are using striping (AIX only) or multiple adapters (AIX or Linux), any window that is allocated on an adapter will prevent other tasks from using windows on that same adapter.
If the adapter use is "Shared":	Only your program tasks have access to the node's CPU, but other program's tasks can share the adapter.	Both the adapter and CPU can be used by a number of your program's tasks and other users.

Note:

1. When using LoadLeveler, the User Space communication subsystem library does not require dedicated use of the IBM High Performance Switch on the node. Adapter use will be defaulted, as in Table 11 on page 25, but shared usage may be specified.
2. Adapter/CPU usage specification is only enforced for jobs using LoadLeveler for node allocation.

Generating an output host list file:

When running parallel programs using LoadLeveler, you can generate an output host list file of the nodes that LoadLeveler allocated. When you have LoadLeveler perform nonspecific node allocation, this enables you to learn which nodes were allocated. This information is vital if you want to perform some postmortem analysis or file cleanup on those nodes, or if you want to rerun the program using the same nodes. To generate a host list file, set the **MP_SAVEHOSTFILE** environment variable to a file name. You can specify this using a relative or full path name. As with most POE environment variables, you can temporarily override the value of **MP_SAVEHOSTFILE** using its associated command line flag **-savehostfile**. Table 13 describes how to set the **MP_SAVEHOSTFILE** environment variable and the **-savehostfile** command line flag.

For example, to save LoadLeveler's node allocation into a file called */u/hinkle/myhosts*, you could:

Table 13. Example of setting the MP_SAVEHOSTFILE environment variable or -savehostfile command line flag

Set the MP_SAVEHOSTFILE environment variable:	Use the -savehostfile flag when invoking the program:
ENTER **export MP_SAVEHOSTFILE=/u/hinkle/myhosts**	**ENTER** **poe** *program* **-savehostfile /u/hinkle/myhosts**

Each record in the output host list file will be the original nonspecific pool request. Following each record will be comments indicating the specific node that was allocated. The specific node is identified by:

- host name
- external IP address

- adapter IP address (which may be the same as the external IP address)

For example, say the input host list file contains the following records:

```
@mypool
@mypool
@mypool
```

The following is a representation of the output host list file.

```
host1_name
! 9.117.11.47               9.117.8.53

!@mypool
host1_name
! 9.117.11.47               9.117.8.53

!@mypool
host1_name
! 9.117.11.47               9.117.8.53

!@mypool
```

Note: The name of your output host list file can be the same as your input host list file. If a file of the same name already exists, it is overwritten by the output host list file.

Step 3c: Set the MP_HOSTFILE environment variable

Use Table 14 to determine if you need to set the **MP_HOSTFILE** environment variable.

Table 14. When to set the MP_HOSTFILE environment variable

You need to set the MP_HOSTFILE environment variable if:	You do not need to set the MP_HOSTFILE environment variable if:
• you are using a host list file other than the default *./host.list* • you are requesting nonspecific node allocation without a host list file.	If your host list file is the default *./host.list*

The default host list file used by the Partition Manager to allocate nodes is called *host.list* and is located in your current directory. You can specify a file other than *host.list* by setting the **MP_HOSTFILE** environment variable to the name of a host list file, or by using either the **-hostfile** or **-hfile** flag when invoking the program, as shown in Table 15. In either case, you can specify the file using its relative or full path name.

For example, say you want to use the host list file *myhosts* located in the directory */u/hinkle*. You could:

Table 15. Example of setting the MP_HOSTFILE environment variable or -hostfile command line flag when using a nondefault host list file

Set the MP_HOSTFILE environment variable:	Use the -hostfile flag when invoking the program:
ENTER **export MP_HOSTFILE=/u/hinkle/myhosts**	**ENTER** **poe** *program* **-hostfile /u/hinkle/myhosts** or **poe** *program* **-hfile /u/hinkle/myhosts**

If you are using LoadLeveler for nonspecific node allocation from a single pool specified by **MP_RMPOOL**, and a host list file exists in the current directory, you must set **MP_HOSTFILE** to an empty string or to the word *NULL*, as shown in Table 16. Otherwise the Partition Manager uses the host list file. You can either:

Table 16. Setting the MP_HOSTFILE environment variable or -hostfile command line flag when requesting nonspecific node allocation without a host list file

Set the MP_HOSTFILE environment variable:	Use the -hostfile flag when invoking the program:
ENTER **export MP_HOSTFILE=** or **export MP_HOSTFILE=**"" or **export MP_HOSTFILE=***NULL*	ENTER **poe** *program* ***-hostfile*** "" or **poe** *program* ***-hostfile*** *NULL*

Step 3d: Set the MP_RESD environment variable

To indicate whether or not LoadLeveler should be used to allocate nodes, you set the **MP_RESD** environment variable to *yes* or *no*. As specified in "Step 3: Set up the execution environment" on page 15, **MP_RESD** controls whether or not the Partition Manager connects to LoadLeveler to allocate processor nodes.

If you are allocating nodes that are **not** part of a LoadLeveler cluster, **MP_RESD** should be set to *no*. If **MP_RESD** is set to *yes*, only nodes within the LoadLeveler cluster can be allocated.

If you are using PE for AIX and you are allocating nodes of an IBM Power Systems network cluster, you do not have LoadLeveler and therefore should set **MP_RESD** to *no*. If you are using a mixed system, you may set **MP_RESD** to *yes*. However, LoadLeveler only has knowledge of nodes that are part of the LoadLeveler cluster. If the additional IBM Power Systems processors are not part of the LoadLeveler cluster, you must also use a host list file to allocate them, and cannot set **MP_RESD** to *yes* in that case.

If you are using PE for Linux and you are not using LoadLeveler, you should set **MP_RESD** to *no* (or allow it to default to *no*). However, LoadLeveler only has knowledge of nodes that are part of the LoadLeveler cluster. If nodes are not part of the LoadLeveler cluster, you must also use a host list file to allocate them, and cannot set **MP_RESD** to *yes* in that case.

As with most POE environment variables, you can temporarily override the value of **MP_RESD** using its associated command line flag **-resd**. Table 17 on page 29 describes how to set the **MP_RESD** environment variable and the **-resd** command line flag.

For example, to specify that you want the Partition Manager to connect LoadLeveler to allocate nodes, you could:

Table 17. Example of setting the MP_RESD environment variable or -resd command line flag

Set the MP_RESD environment variable:	Use the -resd flag when invoking the program:
ENTER **export MP_RESD=***yes*	**ENTER** **poe** *program* **-resd** *yes*

You can also set **MP_RESD** to an empty string. If set to an empty string, or if not set, the default value of **MP_RESD** is interpreted as *yes* or *no* depending on the context. Specifically, the value of **MP_RESD** will be determined by the value of **MP_EUILIB** and whether or not you are using a host list file. Table 18 shows how the context determines the value of **MP_RESD**.

Table 18. How the value of MP_RESD is interpreted

MP_EUILIB setting	and you are using a host list file:	and you are not using a host list file:
If MP_EUILIB is set to ip, an empty string, the word ″NULL″, or if not set:	**MP_RESD** is interpreted as *no* by default, unless: • the host list file includes pool requests, or • the **MP_LLFILE** environment variable is set (or the **-llfile** command line flag is used).	**MP_RESD** is interpreted as *yes* by default.
If MP_EUILIB is set to us:	**MP_RESD** is interpreted as *yes* by default.	**MP_RESD** is interpreted as *yes* by default.

Note: When running POE from a workstation that is external to the LoadLeveler cluster, the **LoadL.so** file set (AIX) or **LoadL-so-***Linux_identifer***.rpm** (Linux) must be installed on the external node (see *Tivoli Workload Scheduler LoadLeveler: Using and Administering* and *IBM Parallel Environment: Installation* for more information).

Step 3e: Set the MP_EUILIB environment variable

During execution, the tasks of your program can communicate via calls to message passing routines. The message passing routines in turn call communication subsystem routines which enable the processor nodes to exchange the message data. Before you invoke your program, you need to decide which communication subsystem implementation you wish to use – the Internet Protocol (IP) communication subsystem or the User Space communication subsystem.

- **The IP communication subsystem implementation** uses the Internet Protocol for communication among processor nodes. You must use the IP communication subsystem if you are using Linux and you do not have a high performance interconnect for which PE provides a User Space protocol.
- **The User Space communication subsystem implementation** uses the User Space protocol in conjunction with a high performance communication adapter. It allows you to drive the communication adapter directly from your parallel tasks rather than via the kernel. You can only use User Space communication when running on a system for which PE supports a User Space protocol.

The **MP_EUILIB** environment variable, or its associated command line flag **-euilib**, is used to indicate which communication subsystem implementation you are using. POE needs to know which communication subsystem implementation to dynamically link in as part of your executable when you invoke it. If you want the IP communication subsystem, **MP_EUILIB** or **-euilib** should specify *ip*. If you want the User Space communication subsystem, **MP_EUILIB** or **-euilib** should specify *us*. In either case, the specification is case-sensitive. Table 19 describes how to set the **MP_EUILIB** environment variable and the **-euilib** command line flag.

For example, say you want to dynamically link in the communication subsystem at execution time. You could:

Table 19. Example of setting the MP_EUILIB environment variable or -euilib command line flag

Set the MP_EUILIB environment variable:	Use the -euilib flag when invoking the program:
ENTER **export MP_EUILIB=ip or us**	**ENTER** **poe** *program* **-euilib ip or us**

Note:

For AIX users, when you invoke a parallel program, your Partition Manager looks to the directory **/usr/lpp/ppe.poe/lib** for the message passing interface and the communication subsystem implementation. If you are running on an IBM Power Systems network cluster, this is the actual location of the message passing interface library. Consult your system administrator for the actual location of the message passing library if necessary.

For Linux users, when you invoke a 32-bit parallel program, your Partition Manager looks to the directory **/opt/ibmhpc/ppe.poe/lib/libmpi** for the message passing interface and the communication subsystem implementation. When you invoke a 64-bit parallel program, your Partition Manager looks to the directory **/opt/ibmhpc/ppe.poe/lib/libmpi64** for the message passing interface and the communication subsystem implementation. Libraries in these two directories are also individually linked into either **/usr/lib** or **/usr/lib64**.

You can make POE look to a directory other than **/usr/lpp/ppe.poe/lib** (AIX) or **/opt/ibmhpc/ppe.poe/lib/libmpi** (Linux) by setting the **MP_EUILIBPATH** environment variable or its associated command line flag **-euilibpath**. This is useful when you get an emergency fix (eFix) library and want to try it out before installing it. Copy the eFix library into a directory and set **MP_EUILIBPATH** to point to it. Table 20 describes how to set the **MP_EUILIBPATH** environment variable and the **-euilibpath** command line flag.

For example, say the communication subsystem library implementations were moved to **/usr/altlib**. To instruct the Partition Manager to look there, you could:

Table 20. Example of setting the MP_EUILIBPATH environment variable or -euilibpath command line flag

Set the MP_EUILIBPATH environment variable:	Use the -euilibpath flag when invoking the program:
ENTER **export MP_EUILIBPATH=/usr/altlib**	**ENTER** **poe** *program* **-euilibpath /usr/altlib**

Step 3f: Set the MP_EUIDEVICE environment variable

Use Table 21 on page 31 to determine if you need to set the **MP_EUIDEVICE** environment variable.

Table 21. When to set the MP_EUIDEVICE environment variable

You need to set the MP_EUIDEVICE environment variable if:	**You do not need to set the MP_EUIDEVICE environment variable if:**
you have set the **MP_EUILIB** environment variable to *ip*, and are using LoadLeveler for node allocation.	you have set the **MP_EUILIB** environment variable to *us*. The Partition Manager assumes that **MP_EUIDEVICE** is *sn_all*. When **MP_EUIDEVICE** is not set, POE uses *sn_all* for both AIX and Linux.

If you are using LoadLeveler, you can specify which adapter to use for message passing for IP using one of the adapter device names or network types configured by LoadLeveler. For User Space (using the IBM High Performance Switch only), you can specify *sn_single* (one window per task) or *sn_all* (multiple windows per tasks).

The **MP_EUIDEVICE** environment variable and its associated command line flag **-euidevice** are used to select an alternate adapter set for communication among processor nodes. If neither **MP_EUIDEVICE** device nor the **-euidevice** flag is set for IP, the communication subsystem library uses the external IP address of each remote node. Table 22 shows the possible, case-sensitive, settings for **MP_EUIDEVICE**.

Table 22. Settings for MP_EUIDEVICE

Setting the MP_EUIDEVICE environment variable to:	**Selects:**
Adapter device name	An adapter device name or network type configured by LoadLeveler.
sn_single	Specify one User Space window per task.
sn_all	Specify multiple windows per task.

Table 23 describes how to set the **MP_EUIDEVICE** environment variable and the **-euidevice** command line flag.

For example, to specify the IBM High Performance Switch, you could:

Table 23. Example of setting the MP_EUIDEVICE environment variable or -euidevice command line flag

Set the MP_EUIDEVICE environment variable:	**Use the -euidevice flag when invoking the program:**
ENTER **export MP_EUIDEVICE=sn_single**	**ENTER** **poe** *program* **-euidevice sn_single**

Note:

1. If you do not set the **MP_EUIDEVICE** environment variable, the default is the adapter set used as the external network address for IP, and for User Space the default is *sn_all*.
2. If using LoadLeveler for node allocation, the adapters must be configured in LoadLeveler. See *Tivoli Workload Scheduler LoadLeveler: Using and Administering* for more information.
3. With PE for AIX, existing User Space applications that set **MP_EUIDEVICE/-euidevice** to *sn_single* on systems using multiple adapters and multiple networks will not benefit from the performance improvements provided by using the *sn_all* value. In this case, you may want to change the **MP_EUIDEVICE/-euidevice** settings for such

applications. Note that User Space applications can set **MP_EUIDEVICE**/**-euidevice** to *sn_single* on systems with multiple adapters and a single network.

4. With PE for Linux, if **MP_EUILIB** is set or defaulted to *ip*, you should not set **MP_EUIDEVICE** or **-euidevice** to *sn_all*. This is because failover and recovery are not supported on Linux with InfiniBand interconnects when using the IP communications subsystem.

Step 3g: Set the MP_DEVTYPE environment variable

Use Table 24 to determine if you need to set the **MP_DEVTYPE** environment variable.

Table 24. When to set the MP_DEVTYPE environment variable

You need to set the MP_DEVTYPE environment variable if:	You do not need to set the MP_DEVTYPE environment variable if:
you want to use the InfiniBand interconnect.	you do not want to use the InfiniBand interconnect or you have set the **MP_EUILIB** environment variable to *us*.

Note: At this time, InfiniBand is the only device type that may be specified with the **MP_DEVTYPE** environment variable.

To use the InfiniBand interconnect, set the **MP_DEVTYPE** environment variable to *ib*, as shown in Table 25. Note that the specification is case-sensitive. For example, say you want to use the InfiniBand interconnect. You could:

Table 25. Example of setting the MP_DEVTYPE environment variable or -devtype command line flag

Set the MP_DEVTYPE environment variable:	Use the -devtype flag when invoking the program:
ENTER **export MP_DEVTYPE=ib**	**ENTER** **poe** *program* **-devtype ib**

If **MP_EUILIB** is set or defaulted to IP when running on the InfiniBand interconnect, the **MP_DEVTYPE** value must be set to *ib*. Otherwise POE uses the host IP address based on the host list file entries, which could be either an IP address or the InfiniBand adapter address, resulting in less than optimum performance.

Step 3h: Set the MP_MSG_API environment variable

The **MP_MSG_API** environment variable, or its associated command line option, is used to indicate to POE which message passing API is being used by a parallel job. Use Table 26 to determine if you need to set the **MP_MSG_API** environment variable.

Table 26. When to set the MP_MSG_API environment variable

You need to set the MP_MSG_API environment variable if:	You do not need to set the MP_MSG_API environment variable if:
A parallel job is using LAPI alone or in conjunction with MPI.	A parallel job is using MPI only.

Step 3i: Set the MP_RMPOOL environment variable

Use Table 27 to determine if you need to set the **MP_RMPOOL** environment variable.

Table 27. When to set the MP_RMPOOL environment variable

You need to set the MP_RMPOOL environment variable if:	You do not need to set the MP_RMPOOL environment variable if:
You are allocating nodes using LoadLeveler and want nonspecific node allocation from a single pool.	You are allocating nodes using a host list file.

After installation of a LoadLeveler cluster, your system administrator divides its processor nodes into a number of pools. Each pool has an identifying pool name or number. When using LoadLeveler, and you want nonspecific node allocation from a single pool, you need to set the **MP_RMPOOL** environment variable to the name or number of that pool. If the value of the **MP_RMPOOL** environment variable is numeric, that pool number must be configured in LoadLeveler. If the value of **MP_RMPOOL** contains any nonnumeric characters, that pool name must be configured as a *feature* in LoadLeveler.

If you need information about available LoadLeveler pools, use the command **llstatus**. To use **llstatus** on a workstation that is external to the LoadLeveler cluster, the following must be installed on the external node:

- For AIX: **LoadL.so** fileset
- For Linux: **LoadL-so-***linux_identifer***.rpm**

See *Tivoli Workload Scheduler LoadLeveler: Using and Administering* and *IBM Parallel Environment: Installation* for more information.

ENTER

llstatus -l (lower case L)

LoadLeveler lists information about all LoadLeveler pools and/or features.

For more information on the **llstatus** command and on LoadLeveler pools, refer to *Tivoli Workload Scheduler LoadLeveler: Using and Administering*.

As with most POE environment variables, you can temporarily override the value of **MP_RMPOOL** using its associated command line flag **-rmpool**. Table 28 describes how to set the **MP_RMPOOL** environment variable and the **-rmpool** command line flag.

For example, to specify pool 6 you could:

Table 28. Example of setting the MP_RMPOOL environment variable or -rmpool command line flag

Set the MP_RMPOOL environment variable:	Use the -rmpool flag when invoking the program:
ENTER **export MP_RMPOOL=6**	**ENTER** **poe** *program* **-rmpool 6**

For additional control over how LoadLeveler allocates nodes within the pool specified by **MP_RMPOOL** or **-rmpool**, you can use the **MP_NODES** or **MP_TASKS_PER_NODE** environment variables or their associated command line options, as shown in Table 29 on page 34.

- The **MP_NODES** and **MP_TASKS_PER_NODE** settings are ignored unless **MP_RMPOOL** is set and no host file is used. A restarted job may actually use these previously ignored settings if **MP_RMPOOL** is used when restarting. See the **poerestart** man page in Chapter 6, "Parallel Environment commands," on page 129 for more information.
- **MP_NODES** or **-nodes** specifies the number of physical nodes on which to run the parallel tasks. You may use it alone or in conjunction with **-tasks_per_node** and/or **-procs**, as described in Table 29, below.
- **MP_TASKS_PER_NODE** or **-tasks_per_node** specifies the number of tasks to be run on each of the physical nodes. You may use it in conjunction with **-nodes** and/or **-procs**, as described in Table 29, but may not use it alone.

Table 29. LoadLeveler node allocation

MP_PROCS set?	MP_TASKS_PER_NODE set?	MP_NODES set?	Conditions and Results
Yes	Yes	Yes	**MP_TASKS_PER_NODE** multiplied by **MP_NODES** must equal **MP_PROCS**, otherwise an error occurs.
Yes	Yes	No	**MP_TASKS_PER_NODE** must divide evenly into **MP_PROCS**, otherwise an error occurs.
Yes	No	Yes	Tasks **0..m-1** are allocated to the first node, tasks **m..2m-1** are allocated to the second node, and so on, where **m** is **MP_PROCS**/**MP_NODES** rounded up.
Yes	No	No	The parallel job will run with the indicated number of **MP_PROCS** (*p*) on *p* nodes.
No	Yes	Yes	The parallel job will consist of **MP_TASKS_PER_NODE** multiplied by **MP_NODES** tasks.
No	Yes	No	An error occurs. **MP_NODES** or **MP_PROCS** *must* be specified with **MP_TASKS_PER_NODE**.
No	No	Yes	One parallel task will be run on each of *n* nodes.
No	No	No	One parallel task will be run on one node.

Note: The examples in Table 29, use the environment variable setting to illustrate each of the three options. The associated command line options may also be used.

Step 4: Invoke the executable

Note:

In order to perform this step, you need to have a user account on, and be able to remotely login to, each of the processor nodes. In addition, each user account must be properly authorized based on the security methods configured by the system administrator. Refer to "POE user authorization" on page 53 for specific details.

The **poe** command enables you to load and execute programs on remote nodes. You can use it to:

- load and execute an SPMD program onto all nodes of your partition. For more information, see "Invoking an SPMD program" on page 36.
- individually load the nodes of your partition. This capability is intended for MPMD programs. For more information, see "Invoking an MPMD program" on page 36.
- load and execute a series of SPMD or MPMD programs, in individual job steps, on the same partition. For more information, see "Loading a series of programs as job steps" on page 38.
- run nonparallel programs on remote nodes. For more information, see "Invoking a nonparallel program on remote nodes" on page 41.

When you invoke **poe**, the Partition Manager allocates processor nodes for each task and initializes the local environment. It then loads your program, and reproduces your local environment, on each processor node. The Partition Manager also passes the option list to each remote node. If your program is in a shared file system, the Partition Manager loads a copy of it on each node. If your program is in a private file system, you will have already manually copied your executable to the nodes as described in "Step 2: Copy files to individual nodes" on page 15. When you are using the message passing interface, the appropriate communication subsystem library implementation (IP or US) is automatically loaded at this time.

Since the Partition Manager attempts to reproduce your local environment on each remote node, your current directory is important. When you invoke **poe**, the Partition Manager will, immediately before running your executable, issue the **cd** command to your current working directory on each remote node. If you are in a local directory that does not exist on remote nodes, you will get an error as the Partition Manager attempts to change to that directory on remote nodes. Typically, this will happen when you invoke **poe** from a directory under **/tmp**. We suggest that you invoke **poe** from a file system that is mounted across the system. If it is important that the current directory be under **/tmp**, make sure that directory exists on all the remote nodes. If you are an AIX user and are running in the C shell, see "Running programs under the C shell (PE for AIX only)" on page 93.

Note: The Parallel Environment opens several file descriptors before passing control to the user. The Parallel Environment will not assign *specific* file descriptors other than standard in, standard out, and standard error.

Before using the **poe** command, you can first specify which programming model you are using by setting the **MP_PGMMODEL** environment variable to either *spmd* or *mpmd*. As with most POE environment variables, you can temporarily override the value of **MP_PGMMODEL** using its associated command line flag **-pgmmodel**. Table 30 describes how to set the **MP_PGMMODEL** environment variable and the **-pgmmodel** command line flag.

For example, if you want to run an MPMD program, you could:

Table 30. Example of setting the MP_PGMMODEL environment variable or -pgmmodel command line flag

Set the MP_PGMMODEL environment variable:	Use the -pgmmodel flag when invoking the program:
ENTER **export MP_PGMMODEL=***mpmd*	**ENTER** **poe** *program* **-pgmmodel** *mpmd*

If you do not set the **MP_PGMMODEL** environment variable or **-pgmmodel** flag, the default programming model is SPMD.

Note: If you load your executable from a mounted file system, you may experience an initial delay while the program is being initialized on all nodes. You may experience this delay even after the program begins executing, because individual pages of the program are brought in on demand. This is particularly apparent during initialization of a parallel application; since individual nodes are synchronized, there are simultaneous demands on the network file transfer system. You can minimize this delay by copying the executable to a local file system on each node, using the **mcp** command.

Invoking an SPMD program

If you have an SPMD program, you want to load the same executable for each task on all nodes of your partition. To do this, follow the **poe** command with the program name and any options. The options can be program options or any of the POE command line flags shown in Chapter 7, "POE Environment variables and command line flags," on page 215. You can also invoke an SPMD program by entering the program name and any options:

ENTER
: **poe** *program [options]*

  or

  program [options]

You can also enter **poe** without a program name:

ENTER
: **poe** *[options]*

  Once your partition is established, a prompt appears.

ENTER
: the name of the program you want to load. You can follow the program name with any program options or a subset of the POE flags.

Note: For National Language Support, POE displays messages located in an externalized message catalog. POE checks the **LANG** and **NLSPATH** environment variables, and if either is not set, it will set up the following defaults.

For AIX:

- **LANG=C**
- **NLSPATH=/usr/lib/nls/msg/%L/%N**

For Linux:

- **LANG=en_US**
- **NLSPATH=/usr/share/locale/%L/%N**

Invoking an MPMD program

Note: You must set the **MP_PGMMODEL** environment variable or **-pgmmodel** flag to invoke an MPMD program.

With an SPMD application, the name of the same executable is sent to, and runs as each task on all of the processor nodes of your partition. If you are invoking an

MPMD application, you are dealing with more than one program and need to individually specify the executable to be run for each task of your partition.

For example, say you have two programs – *master* and *workers* – designed to run together and communicate via calls to message passing subroutines. The program *master* is designed to run as one task, perhaps task zero. The *workers* program is designed to run as separate tasks on any number of other nodes, and each task knows it is to take direction from task zero. The *master* program will coordinate and synchronize the execution of all the worker tasks. Neither program can run without the other.

You can establish a partition and load each node individually using:
- standard input (from the keyboard or redirected)
- a POE commands file

Loading nodes individually from standard input:

To establish a partition and load each node individually using STDIN:

ENTER
poe *[options]*

The Partition Manager allocates the processor nodes of your partition. Once your partition is established, a prompt containing both the logical task identifier 0 and the actual host name to which it maps, appears.

ENTER
the name of the program you want to load on task 0. You can follow the program name with any program options or a subset of the POE flags.

A prompt for the next task number in the partition displays.

ENTER
the name of the program you want to load as each task, as you are prompted.

When you have specified the program to run as the last task of your partition, the message “Partition loaded...” displays and execution begins.

For additional illustration, the following shows the command prompts that would appear, as well as the program names you would enter, to load the example *master* and *workers* programs. This example assumes that the **MP_PROCS** environment variable is set to 5, and that you wish to run 2 worker tasks per node, and the master on a node by itself. Your host list file would list host1_name once, but host2_name and host3_name twice each.

```
% poe

0:host1_name> master [options]

1:host2_name> workers [options]

2:host2_name> workers [options]

3:host3_name> workers [options]

4:host3_name> workers [options]

Partition loaded...
```

Note: You can use the following POE command line flags on individual program names, but not those that are used to set up the partition.

- **-infolevel** or **-ilevel**

Loading nodes individually using a POE commands file:

The **MP_CMDFILE** environment variable, and its associated command line flag **-cmdfile**, let you specify the name of a POE commands file. You can use such a file when individually loading a partition – thus freeing STDIN. The POE commands file simply lists the individual programs you want to load and run on the nodes of your partition. The programs are loaded in task order. For example, say you have a typical master/workers MPMD program that you want to run as 5 tasks. Your POE commands file would contain:

```
master [options]

workers [options]

workers [options]

workers [options]

workers [options]
```

Once you have created a POE commands file, you can specify it using a relative or full path name on the **MP_CMDFILE** environment variable or **-cmdfile** flag. Table 31 describes how to set the **MP_CMDFILE** environment variable and the **-cmdfile** command line flag.

For example, if your POE commands file is **/u/hinkle/mpmdprog**, you could:

Table 31. Example of setting the MP_CMDFILE environment variable or -cmdfile command line flag

Set the MP_CMDFILE environment variable:	Use the -cmdfile flag on the poe command:
ENTER **export MP_CMDFILE=/u/hinkle/mpmdprog**	**ENTER** **poe -cmdfile /u/hinkle/mpmdprog**

Once you have set the **MP_CMDFILE** environment variable to the name of the POE commands file, you can individually load the nodes of your partition. To do this:

ENTER
: **poe** *[options]*

The Partition Manager allocates the processor nodes of your partition. The programs listed in your POE commands file are run on the nodes of your partition.

Loading a series of programs as job steps

By default, the Partition Manager releases your partition when your program completes its run. However, you can set the environment variable **MP_NEWJOB**, or its associated command line flag **-newjob**, to specify that the Partition Manager should maintain your partition for multiple job steps. Table 32 on page 39 describes how to set the **MP_SAVEHOSTFILE** environment variable and the **-savehostfile** command line flag.

For example, say you have three separate SPMD programs. The first one sets up a particular computation by adding some files to **/tmp** on each of the processor nodes on the partition. The second program does the actual computation. The third program does some postmortem analysis and file cleanup. These three parallel programs must run as job steps on the same processor nodes in order to work correctly. While specific node allocation using a host list file might work, the requested nodes might not be available when you invoke each program. The better solution is to instruct the Partition Manager to maintain your partition after execution of each program completes. You can then read multiple job steps from:

- standard input
- a POE commands file using the **MP_CMDFILE** environment variable.

In either case, you must first specify that you want the Partition Manager to maintain your partition for multiple job steps. To do this, you could:

Table 32. Example of setting the MP_NEWJOB environment variable or -newjob command line flag

Set the MP_NEWJOB environment variable:	**Use the -newjob flag on the poe command:**
ENTER **export MP_NEWJOB=***yes*	**ENTER** **poe -newjob** *yes*

Note:

1. **poe** is its own shell. Whether successive steps run after a step completes is a function of the exit code, as described in *IBM Parallel Environment: MPI Programming Guide*

Reading job steps from standard input:

Say you want to run three SPMD programs – *setup*, *computation*, and *cleanup* – as job steps on the same partition. Assuming STDIN is keyboard entry, **MP_PGMMODEL** is set to *spmd*, and **MP_NEWJOB** is set to *yes*, you would:

ENTER
: **poe** [*poe-options*]

The Partition Manager allocates the processor nodes of your partition, and the following prompt displays:

```
0031-503 Enter program name (or quit):
```

ENTER
: *setup* [*program-options*]

The program *setup* executes on all nodes of your partition. When execution completes, the following prompt displays:

```
0031-503 Enter program name (or quit):
```

ENTER
: *computation* [*program-options*]

The program *computation* executes on all nodes of your partition. When execution completes, the following prompt displays:

```
0031-503 Enter program name (or quit):
```

ENTER
: *cleanup* [*program-options*]

The program *cleanup* executes on all nodes of your partition. When execution completes, the following prompt displays:

```
0031-503 Enter program name (or quit):
```

ENTER

quit

or

<**Ctrl-d**>

The Partition Manager releases the nodes of your partition.

Note:

1. You can also run a series of MPMD programs in job step fashion from STDIN. If **MP_PGMMODEL** is set to *mpmd*, the Partition Manager will, after each step completes, prompt you to individually reload the partition as described in "Loading nodes individually from standard input" on page 37.
2. When **MP_NEWJOB** is *yes*, the Partition Manager, by default, looks to STDIN for job steps. However, if the environment variable **MP_CMDFILE** is set to the name of a POE commands file as described in "Reading job steps from a POE commands file," the Partition Manger will look to the commands file instead. To ensure that job steps are read from STDIN, check that the **MP_CMDFILE** environment variable is unspecified.

Multi-step STDIN for newjob mode:

POE's STDIN processing model allows redirected STDIN to be passed to all steps of a newjob sequence, when the redirection is from a file. If redirection is from a pipe, POE does not distribute the input to each step, only to the first step.

Reading job steps from a POE commands file:

The **MP_CMDFILE** environment variable, and its associated command line flag **-cmdfile**, lets you specify the name of a POE commands file. If **MP_NEWJOB** is *yes*, you can have the Partition Manager read job steps from a POE commands file. The commands file in this case simply lists the programs you want to run as job steps. For example, say you want to run the three SPMD programs *setup*, *computation*, and *cleanup* as job steps on the same partition. Your POE commands file would contain the following three lines:

```
setup [program-options]

computation [program-options]

cleanup [program-options]
```

`Program-options` represent the actual values you need to specify.

If you are loading a series of MPMD programs, the POE commands file is also responsible for individually loading the partition. For example, say you had three master/worker MPMD job steps that you wanted to run as 4 tasks on the same partition. The following is a representation of what your POE commands file would contain. `Options` represent the actual values you need to specify.

```
master1 [options]

workers1 [options]
```

```
workers1 [options]

workers1 [options]

master2 [options]

workers2 [options]

workers2 [options]

workers2 [options]

master3 [options]

workers3 [options]

workers3 [options]

workers3 [options]
```

While you could also redirect STDIN to read job steps from a file, a POE commands file gives you more flexibility by not tying up STDIN. You can specify a POE commands file using its relative or full path name.

Table 33 provides an example of specifying a POE commands file. Say your POE commands file is called */u/hinkle/jobsteps*. To specify that the Partition Manager should read job steps from this file rather than STDIN, you could:

Table 33. Example of specifying a POE commands file from which the Partition Manager should read job steps

Set the MP_CMDFILE environment variable:	Use the -cmdfile flag on the poe command:
ENTER **export MP_CMDFILE=/u/hinkle/jobsteps**	**ENTER** **poe -cmdfile /u/hinkle/jobsteps**

Once **MP_NEWJOB** is set to *yes*, and **MP_CMDFILE** is set to the name of your POE commands file, you would:

ENTER
poe [*poe-options*]

The Partition Manager allocates the processor nodes of your partition, and reads job steps from your POE commands file. The Partition Manager does not release your partition until it reaches the end of your commands file.

Invoking a nonparallel program on remote nodes

You can also use POE to run nonparallel programs on the remote nodes of your partition. Any executable (binary file, shell script, UNIX, AIX, or Linux utility) is suitable, and it does not need to have been compiled with **mpcc_r**, **mpCC_r**, or **mpxlf_r** (AIX) or **mpcc**, **mpCC**, or **mpfort** (Linux). For example, if you wanted to check the process status (using the AIX or Linux command **ps**) for all remote nodes in your partition, you would:

ENTER
poe ps

The process status for each remote node is written to standard out (STDOUT) at your home node. How STDOUT from all the remote nodes is handled at your home node depends on the output mode. See "Managing standard output (STDOUT)" on page 48 for more information.

Controlling program execution

There are a number of additional POE environment variables for monitoring and controlling program execution.

- **MP_EUIDEVELOP** environment variable to specify that you want to run your program in message passing develop mode. In this mode, more detailed checking of your program is performed.
- **MP_RETRY** environment variable to make POE wait for processor nodes to become available.
- **MP_RETRYCOUNT** environment variable to specify the number of times the Partition Manager should request nodes before returning.
- **MP_NOARGLIST** and **MP_FENCE** environment variable to make POE ignore arguments.
- **MP_STDINMODE** environment variable to manage standard input.
- **MP_STDOUTMODE** environment variable to manage standard output.
- **MP_LABELIO** environment variable to label message output with task identifiers.
- **MP_INFOLEVEL** environment variable to specify the level of messages you want reported to standard error.
- **MP_PMDLOG** environment variable to generate a diagnostic log on remote nodes.
- **MP_IONODEFILE** environment variable to specify an I/O node file that indicates which nodes should participate in parallel I/O.
- **MP_CKPTFILE** (PE for AIX only) environment variable to define the base name of the checkpoint file when checkpointing a program. See “Checkpointing and restarting programs (PE for AIX only)” on page 54 for more information.
- **MP_CKPTDIR** (PE for AIX only) environment variable to define the directory where the checkpoint file will reside when checkpointing a program. See “Checkpointing and restarting programs (PE for AIX only)” on page 54 for more information.
- **MP_TASK_AFFINITY** environment variable to attach each task of a parallel job to one of the system resource sets (*rsets* for AIX or *CPU sets* for Linux), at the Multi-chip Module (MCM), core, or CPU level. See “Managing task affinity on large SMP nodes” on page 57 for more information.

For a complete listing of all POE environment variables, see Chapter 7, “POE Environment variables and command line flags,” on page 215.

Specifying develop mode

You can run programs in one of two modes – *develop mode* or *run mode*. In develop mode, intended for developing applications, the message passing interface performs more detailed checking during execution. Because of the additional checking it performs, develop mode can significantly slow program performance. In run mode, intended for completed applications, only minimal checking is done. While run mode is the default, you can use the **MP_EUIDEVELOP** environment variable to specify message passing develop mode.

As with most POE environment variables, **MP_EUIDEVELOP** has an associated command line flag **-euidevelop**. Table 34 on page 43 describes how to set the **MP_EUIDEVELOP** environment variable and the **-euidevelop** command line flag.

For example, to specify MPI develop mode, you could:

Table 34. Example of setting the MP_EUIDEVELOP environment variable or -euidevelop command line flag

Set the MP_EUIDEVELOP environment variable:	Use the -euidevelop flag when invoking the program:
ENTER **export MP_EUIDEVELOP=***yes*	**ENTER** **poe** *program* **-euidevelop** *yes*

You could also specify debug develop mode by setting **MP_EUIDEVELOP** to *deb*.

To later go back to run mode, set **MP_EUIDEVELOP** to *no*.

To further limit parameter checking, set **MP_EUIDEVELOP** to *min*, for *minimum*. Programs with errors may fail in unpredictable ways.

Making POE wait for processor nodes

If you are using Loadleveler, and there are not enough available nodes to run your program, the Partition Manager, by default, returns immediately with an error. Your program does not run. Using the **MP_RETRY** and **MP_RETRYCOUNT** environment variables, however, you can instruct the Partition Manager to repeat the node request a set number of times at set intervals. Each time the Partition Manager repeats the node request, it displays the following message:

```
Retry allocation     ......press control-C to terminate
```

The **MP_RETRY** environment variable, and its associated command line flag **-retry**, specifies the interval (in seconds) to wait before repeating the node request. The **MP_RETRYCOUNT** environment variable, and its associated command line flag **-retrycount**, specifies the number of times the Partition Manager should make the request before returning. Table 35 describes how to set the **MP_RETRY** and **MP_RETRYCOUNT** environment variables and the **-retry** and **-retrycount** command line flags.

For example, if you wanted to retry the node request five times at five minute (300 second) intervals, you could:

Table 35. Example of setting the MP_RETRY and MP_RETRYCOUNT environment variables or -retry and -retrycount command line flags

Set the MP_RETRY and MP_RETRYCOUNT environment variables:	Use the -retry and -retrycount flags when invoking the program:
ENTER **export MP_RETRY=***300* **export MP_RETRYCOUNT=***5*	**ENTER** **poe** *program* **-retry** *300* **-retrycount** *5*

Note: If the **MP_RETRYCOUNT** environment variable or the **-retrycount** command line flag is used, the **MP_RETRY** environment variable or the **-retry** command line flag must be set to at least one second.

If **MP_RETRY** or **-retry** is set to the character string **wait**, instead of a number, no retries are attempted by POE, and the job remains enqueued in LoadLeveler until LoadLeveler either schedules or cancels the job. **wait** is not case sensitive.

Making POE ignore arguments

When you invoke a parallel executable, you can specify an argument list consisting of a number of program options and POE command line flags. The argument list is parsed by POE – the POE command line flags are removed and the remainder of the list is passed on to the program. If any of your program arguments are identical to POE command line flags, however, this can cause problems. For example, say you have a program that takes the argument **-retry**. You invoke the program with the **-retry** option, but it does not execute correctly. This is because there is also a POE command line flag **-retry**. POE parses the argument list and so the **-retry** option is never passed on to your program. There are two ways to correct this sort of problem. You can:

- make POE ignore the entire argument list using the **MP_NOARGLIST** environment variable.
- make POE ignore a portion of the argument list using the **MP_FENCE** environment variable.

Making POE ignore the entire argument list

When you invoke a parallel executable, POE, by default, parses the argument list and removes all POE command line flags before passing the rest of the list on to the program. Using the environment variable **MP_NOARGLIST**, you can prevent POE from parsing the argument list. To do this:

ENTER

export MP_NOARGLIST=*yes*

When the **MP_NOARGLIST** environment variable is set to *yes*, POE does not examine the argument list at all. It simply passes the entire list on to the program. For this reason, you can not use any POE command line flags, but must use the POE environment variables exclusively. While most POE environment variables have associated command line flags, **MP_NOARGLIST**, for obvious reasons, does not. To specify that POE should again examine argument lists, either set **MP_NOARGLIST** to *no*, or unset it.

ENTER

export MP_NOARGLIST=*no*

or

unset MP_NOARGLIST

Making POE ignore a portion of the argument list

When you invoke a parallel executable, POE, by default, parses the entire argument list and removes all POE command line flags before passing the rest of the list on to the program. You can use a fence, however, to prevent POE from parsing the remainder of the argument list. A *fence* is simply a character string you define using the **MP_FENCE** environment variable. Once defined, you can use the fence to separate those arguments you want parsed by POE from those you do not. For example, say you have a program that takes the argument **-retry**. Because there is also a POE command line flag **-retry**, you need to put this argument after a fence. To do this, you could:

ENTER

export MP_FENCE=*Q*

poe *program* **-procs** *26* **-infolevel** *2 Q -retry RGB*

While this example defines Q as the fence, keep in mind that the fence can be any character string. Any arguments placed after the fence are passed by POE, unexamined, to the program. While most POE environment variables have associated command line flags, **MP_FENCE** does not.

POE argument limits

The maximum length for POE program arguments is 24,576 bytes. This is a fixed limit and cannot be changed. If this limit is exceeded, an error message will be displayed and POE will terminate. The length of the remote program arguments that can be passed on POE's command line is 24,576 bytes minus the number of bytes that are used for POE arguments.

Managing standard input, output, and error

POE lets you control standard input (STDIN), standard output (STDOUT), and standard error (STDERR) in several ways. You can continue using the traditional I/O manipulation techniques such as redirection and piping, and can also:

- determine whether a single task or all parallel tasks should receive data from STDIN.
- determine whether a single task or all parallel tasks should write to STDOUT. If all tasks are writing to STDOUT, you can further define whether or not the messages are ordered by task id.
- specify the level of messages that will be reported to STDERR during program execution.
- specify that messages to STDOUT and STDERR should be labeled by task id.

Managing standard input (STDIN)

STDIN is the primary source of data going into a command. Usually, STDIN refers to keyboard input. If you use redirection or piping, however, STDIN could refer to a file or the output from another command. How you manage STDIN for a parallel application depends on whether or not its parallel tasks require the same input data. Using the environment variable **MP_STDINMODE** or the command line flag **-stdinmode**, you can specify that:

- all tasks should receive the same input data from STDIN. This is *multiple input mode*.
- STDIN should be sent to a single task of your partition. This is *single input mode*.
- no task should receive input data from STDIN.

Multiple input mode:

Setting **MP_STDINMODE** to *all* indicates that all tasks should receive the same input data from STDIN. The home node Partition Manager sends STDIN to each task as it is read.

Table 36 on page 46 describes how to specify multiple input mode with the **MP_STDINMODE** environment variable and the **-stdinmode** command line flag.

To specify multiple input mode, so all tasks receive the same input data from STDIN, you could:

Table 36. Example of specifying multiple input mode with the MP_STDINMODE environment variable or -stdinmode command line flag

Set the MP_STDINMODE environment variable:	Use the -stdinmode flag when invoking the program:
ENTER **export MP_STDINMODE=all**	**ENTER** **poe** *program* **-stdinmode all**

Note: If you do not set the **MP_STDINMODE** environment variable or use the **-stdinmode** command line flag, multiple input mode is the default.

Single input mode:

There are times when you only want a single task to read from STDIN. To do this, you set **MP_STDINMODE** to the appropriate task id. For example, say you have an MPMD application consisting of two programs – *master* and *workers*. The program *master* is designed to run as a single task on one processor node. The *workers* program is designed to run as separate tasks on any number of other nodes. The *master* program handles all I/O, so only its task needs to read STDIN.

Table 37 describes how to specify multiple input mode with the **MP_STDINMODE** environment variable and the **-stdinmode** command line flag.

If *master* is running as task 0, you need to specify that only task 0 should receive STDIN. To do this, you could:

Table 37. Example of specifying single input mode with the MP_STDINMODE environment variable or -stdinmode command line flag

Set the MP_STDINMODE environment variable:	Use the -stdinmode flag when invoking the program:
ENTER **export MP_STDINMODE=0**	**ENTER** **poe** *program* **-stdinmode 0**

Using MP_HOLD_STDIN (PE for AIX only):

Note: Earlier versions of Parallel Environment for AIX required the use of the MP_HOLD_STDIN environment variable in certain cases when redirected STDIN was used. The Parallel Environment components have now been modified to control the STDIN flow internally, so the use of this environment variable is no longer required, and will have no effect on STDIN handling.

Using redirected STDIN:

Note: Wherever the following description refers to a POE environment variable (starting with **MP_**), the use of the associated command line option produces the same effect.

A POE process can use its STDIN in two ways. First, if the program name is not supplied on the command line and no command file (**MP_CMDFILE**) is specified, POE uses STDIN to resolve the names of the programs to be run as the remote tasks. Second, any *remaining* STDIN is then distributed to the remote tasks as indicated by the **MP_STDINMODE** setting. In this dual STDIN model, redirected STDIN can then pose two problems:

1. If using job steps (**MP_NEWJOB=yes**), the *remaining* STDIN is always consumed by the remote tasks during the first job step.
2. If POE attempts program name resolution on the redirected STDIN, program behavior can vary when using job steps, depending on the type of redirection used and the size of the redirected STDIN.

The first problem is addressed in POE by performing a rewind of STDIN between job steps (only if STDIN is redirected from a file, for reasons beyond the scope of these instructions). The second problem is addressed by providing an additional setting for **MP_STDINMODE** of **none**, which tells POE to only use STDIN for program name resolution. As far as STDIN is concerned, **none** ever gets delivered to the remote tasks. This provides an additional method of reliably specifying the program name to POE, by redirecting STDIN from a file or pipe, or by using the shell's here-document syntax in conjunction with the **none** setting. If **MP_STDINMODE** is not set to **none** when POE attempts program name resolution on redirected STDIN, program behavior is undefined.

The following scenarios describe in more detail the effects of using (or not using) an **MP_STDINMODE** of **none** when redirecting (or not redirecting) STDIN, as shown in the example:

		Is STDIN Redirected?	
		Yes	No
Is MP_STDINMODE set to *none*?	Yes	A	B
	No	C	D

Scenario A: POE will use the redirected STDIN for program name resolution, only if no program name is supplied on the command line (**MP_CMDFILE** is ignored when **MP_STDINMODE=none**). No STDIN is distributed to the remote tasks. No rewind of STDIN is performed when **MP_STDINMODE=none**.

Scenario B: POE will use the keyboard STDIN for program name resolution, only if no program name is supplied on the command line (**MP_CMDFILE** is ignored when **MP_STDINMODE=none**). No STDIN is distributed to the remote tasks. No rewind of STDIN is performed when **MP_STDINMODE=none** (also, STDIN is not from a file).

Scenario C: POE will use the redirected STDIN for program name resolution, if required, and will distribute *remaining* STDIN to the remote tasks. If STDIN is intended to be used for program name resolution, program behavior is *undefined* in this case, since POE was not informed of this by setting **STDINMODE** to none (see Problem 2 above). If STDIN is redirected from a file, POE will rewind STDIN between each job step. For large amounts of redirected STDIN (more than 4k bytes), programs should consider bypassing the home node POE binary as described in the *Standard I/O requires special attention* section in *IBM Parallel Environment: MPI Programming Guide*.

Scenario D: POE will use the keyboard STDIN for program name resolution, if required. Any *remaining* STDIN is distributed to the remote tasks. No rewind of STDIN is performed since STDIN is not from a file.

Managing standard output (STDOUT)

STDOUT is where the data coming from the command will eventually go. Usually, STDOUT refers to the display. If you use redirection or piping, however, STDOUT could refer to a file or another command. How you manage STDOUT for a parallel application depends on whether you want output data from one task or all tasks. If all tasks are writing to STDOUT, you can also specify whether or not output is ordered by task id. Using the environment variable **MP_STDOUTMODE**, you can specify that:

- all tasks should write output data to STDOUT asynchronously. This is *unordered output mode*.
- output data from each parallel task should be written to its own buffer, and later all buffers should be flushed, in task order, to STDOUT. This is *ordered output mode*.
- a single task of your partition should write to STDOUT. This is *single output mode*.

Unordered output mode:

Setting **MP_STDOUTMODE** to *unordered* specifies that all tasks should write output data to STDOUT asynchronously.

Table 38 describes how to specify unordered output mode by setting the **MP_STDOUTMODE** environment variable and the **-stdoutmode** command line flag.

To specify unordered output mode, you could:

Table 38. Example of specifying unordered output mode with the MP_STDOUTMODE environment variable or -stdoutmode command line flag

Set the MP_STDOUTMODE environment variable:	Use the -stdoutmode flag when invoking the program:
ENTER **export MP_STDOUTMODE=***unordered*	**ENTER** **poe** *program* **-stdoutmode** *unordered*

Note:

1. If you do not set the **MP_STDOUTMODE** environment variable or use the **-stdoutmode** command line flag, unordered output mode is the default.
2. If you are using unordered output mode, you will probably want the messages labeled by task id. Otherwise it will be difficult to know which task sent which message. See "Labeling message output" on page 50 for more information.
3. You can also specify unordered output mode from your program by calling the **MP_STDOUTMODE** or **mpc_stdoutmode** Parallel Utility Function. Refer to *IBM Parallel Environment: MPI Subroutine Reference* for more information.
4. Although the environment variable and Parallel Utility Function above are both described as ***MP_STDOUTMODE***, they are each used independently for their specific purposes.

Ordered output mode:

Setting **MP_STDOUTMODE** to *ordered* specifies ordered output mode. In this mode, each task writes output data to its own buffer. Later, all the task buffers are flushed, in order of task id, to STDOUT. The buffers are flushed when:

- any one of the individual task buffers fills
- execution of the program completes.
- all tasks explicitly flush the buffers by calling the MP_FLUSH or mpc_flush Parallel Utility Function.
- tasks change output mode using calls to Parallel Utility Functions. For more information on Parallel Utility Functions, refer to *IBM Parallel Environment: MPI Subroutine Reference*

Table 39 describes how to specify ordered output mode by setting the **MP_STDOUTMODE** environment variable and the **-stdoutmode** command line flag.

To specify ordered output mode, you could:

Table 39. Example of specifying ordered output mode with the MP_STDOUTMODE environment variable or -stdoutmode command line flag

Set the MP_STDOUTMODE environment variable:	Use the -stdoutmode flag when invoking the program:
ENTER **export MP_STDOUTMODE=***ordered*	**ENTER** **poe** *program* **-stdoutmode** *ordered*

Note: You can also specify ordered output mode from your program by calling the **MP_STDOUTMODE** or mpc_stdoutmode Parallel Utility Function. Refer to *IBM Parallel Environment: MPI Subroutine Reference* for more information.

Single output mode:

You can specify that only one task should write its output data to STDOUT. To do this, you set **MP_STDOUTMODE** to the appropriate task id. For example, say you have an SPMD application in which all the parallel tasks are sending the exact same output messages. For easier readability, you would prefer output from only one task – task 0.

Table 40 describes how to single output mode by setting the **MP_STDOUTMODE** environment variable and the **-stdoutmode** command line flag.

To specify this, you could:

Table 40. Example of specifying single output mode with the MP_STDOUTMODE environment variable or -stdoutmode command line flag

Set the MP_STDOUTMODE environment variable:	Use the -stdoutmode flag when invoking the program:
ENTER **export MP_STDOUTMODE=***0*	**ENTER** **poe** *program* **-stdoutmode** *0*

Note: You can also specify single output mode from your program by calling the **MP_STDOUTMODE** or mpc_stdoutmode Parallel Utility Function. Refer to *IBM Parallel Environment: MPI Subroutine Reference* for more information.

Labeling message output

You can set the environment variable **MP_LABELIO**, or use the **-labelio** flag when invoking a program, so that output from the parallel tasks of your program are labeled by task id. While not necessary when output is being generated in *single* mode, this ability can be useful in *ordered* and *unordered* modes. For example, say the output mode is *unordered*. You are executing a program and receiving asynchronous output messages from all the tasks. This output is not labeled, so you do not know which task has sent which message. It would be clearer if the unordered output was labeled. For example:

```
 7: Hello World

 0: Hello World

 3: Hello World

23: Hello World

14: Hello World

 9: Hello World
```

Table 41 describes how to set the **MP_LABELIO** environment variable and the **-labelio** command line flag.

To have the messages labeled with the appropriate task id, you could:

Table 41. Example of setting the MP_LABELIO environment variable or -labelio command line flag

Set the MP_LABELIO environment variable:	Use the -labelio flag when invoking the program:
ENTER **export MP_LABELIO=***yes*	**ENTER** **poe** *program* **-labelio** *yes*

To no longer have message output labeled, set the **MP_LABELIO** environment variable to *no*.

Setting the message reporting level for standard error (STDERR)

You can set the environment variable **MP_INFOLEVEL** to specify the level of messages you want from POE. You can set the value of **MP_INFOLEVEL** to one of the integers shown in the following table. The integers *0*, *1*, and *2* give you different levels of informational, warning, and error messages. The integers *3* through *6* indicate debug levels that provide additional debugging and diagnostic information. Should you require help from the IBM Support Center in resolving a PE-related problem, you will probably be asked to run with one of the debug levels. As with most POE environment variables, you can override **MP_INFOLEVEL** when you invoke a program. This is done using either the **-infolevel** or **-ilevel** flag followed by the appropriate integer.

When **MP_INFOLEVEL** is set to 0, the **STDERR** output may contain null characters under conditions where warning or informational messages would be displayed under higher levels.

Table 42 on page 51 shows the valid values for **MP_INFOLEVEL** and the level of message reporting provided by each.

Table 42. MP_INFOLEVEL values and associated levels of message reporting

This integer:	Indicates this level of message reporting:	In other words:
0	Error	Only error messages from POE are written to STDERR.
1	Normal	Warning and error messages from POE are written to STDERR. This level of message reporting is the default.
2	Verbose	Informational, warning, and error messages from POE are written to STDERR.
3	Debug Level 1	Informational, warning, and error messages from POE are written to STDERR. Also written is some high-level debugging and diagnostic information.
4	Debug Level 2	Informational, warning, and error messages from POE are written to STDERR. Also written is some high- and low-level debugging and diagnostic information.
5	Debug Level 3	Debug level 2 messages plus some additional loop detail.
6	Debug Level 4	Debug level 3 messages plus other informational error messages for the greatest amount of diagnostic information.

As an example, say you want the POE message level set to verbose. Table 43 shows the two ways to do this. You could:

Table 43. Example of setting MP_INFOLEVEL to verbose

Set the MP_INFOLEVEL environment variable:	Use the -infolevel flag when invoking the program:
ENTER **export MP_INFOLEVEL**=2	**ENTER** **poe** *program* **-infolevel** 2 or **poe** *program* **-ilevel** 2

As with most POE command line flags, the **-infolevel** or **-ilevel** flag temporarily override their associated environment variable.

Generating a diagnostic log on remote nodes

Using the **MP_PMDLOG** environment variable, you can also specify that diagnostic messages should be logged to a file in **/tmp** on each of the remote nodes of your partition. If you do not wish to store the log file in **/tmp**, you can use the **MP_PMDLOG_DIR** environment variable to specify a different directory.

The log file is named *mplog.jobid.n* where *jobid* is a unique job identifier. The *jobid* will be the same for all remote nodes. Should you require help from the IBM Support Center in resolving a PE-related problem, you will probably be asked to generate these diagnostic logs.

The ability to generate diagnostic logs on each node is particularly useful for isolating the cause of abnormal termination, especially when the connection between the remote node and the home node Partition Manager has been broken. As with most POE environment variables, you can temporarily override the value of **MP_PMDLOG** using its associated command line flag **-pmdlog**.

Table 44 on page 52 describes how to set the **MP_PMDLOG** environment variable and the **-pmdlog** command line flag.

For example, to generate a **pmd** log file, you could:

Table 44. Example of setting the MP_PMDLOG environment variable or -pmdlog command line flag

Set the MP_PMDLOG environment variable:	Use the -pmdlog flag when invoking the program:
ENTER **export MP_PMDLOG=***yes*	**ENTER** **poe** *program* **-pmdlog** *yes*

Note: By default, **MP_PMDLOG** is set to *no*. No diagnostic logs are generated. You should not run **MP_PMDLOG** routinely, because this greatly impacts performance and fills up your file system space.

Determining which nodes will participate in parallel file I/O

MPI has a number of subroutines that enable your application program to perform efficient parallel input-output operations. These subroutines (collectively referred to as *MPI-IO*) allow efficient file I/O on a data structure which is distributed across several tasks for computation, but organized in a unified way in a single underlying file. MPI-IO presupposes a single parallel file system underlying all the tasks in the parallel job; PE's implementation of it is intended for use with the IBM General Parallel File System™ (GPFS™).

If your application program uses MPI-IO subroutines, all tasks in your MPI job will, by default, participate in parallel I/O. You can, however, specify that only tasks on a subset of the nodes in your job should handle parallel I/O. You might want to do this to ensure that all I/O operations are performed on the same node. To specify the nodes that should participate in parallel I/O, you:

- create an *I/O node file* (a text file that lists the nodes that should handle parallel I/O) and
- set the **MP_IONODEFILE** environment variable to the name of the I/O node file. As with most POE environment variables, **MP_IONODEFILE** has an associated command line flag **-ionodefile**.

For example, say your job will be run with the following host list file dictating the nodes on which your program should run.

```
host1_name
host2_name
host3_name
host4_name
host5_name
host6_name
```

Say, however, that you want parallel I/O handled by only two of these nodes — *host5_name* and *host6_name*. To specify this, you would create an I/O node file that lists just the two host names.

```
host5_name
host6_name
```

One situation in which **MP_IONODEFILE** becomes useful is when running on a cluster of workstations which will not have a true parallel file system across multiple machines. By selecting one workstation to do the actual I/O, you can reliably use JFS, NFS, and AFS® files with MPI-IO across multiple machines. (The file systems currently used, like NFS and AFS, to make a set of files available to multiple workstations are not parallel file systems in the way that GPFS is.) With respect to MPI-IO, a cluster without GPFS should use an I/O node file.

There should be no comments or blank lines in the I/O node file, there should be only one node name per line. Node names may be in any form recognizable to name service on the machine. Names which are not recognizable or which appear more that once yield advisory messages. Names which are valid but which do not represent nodes in the job are ignored. If **MP_IONODEFILE** is used and no node listed in the file is involved in the job, the job will abort. **MP_IONODEFILE** is most useful when used in conjunction with a host list file.

To indicate that the Partition Manager should use a particular I/O node file to determine which nodes handle parallel I/O, you must set the **MP_IONODEFILE** environment variable (or use the **-ionodefile** command line flag to specify) the name of the file. Table 45 describes how to set the **MP_IONODEFILE** environment variable and the **-iodnodefile** command line flag. You can specify the file using its relative or full path name.

For example, say you have created an I/O node file **ionodes** in the directory **/u/dlecker**. You could:

Table 45. Example of setting the MP_IONODEFILE environment variable or -ionodefile command line flag

Set the MP_IONODEFILE environment variable:	Use the -ionodefile flag when invoking the program:
ENTER **export MP_IONODEFILE=/u/dlecker/ionodes**	**ENTER** **poe** *program* **-ionodefile /u/dlecker/ionodes**

POE user authorization

Under AIX, PE uses an enhanced set of security methods based on Cluster Security Services in RSCT. You can also use AIX-based user authorization.

Under Linux, PE supports a limited set of user authorization mechanisms. POE uses a configuration option for the system administrator to define the security mechanism which, on Linux, is limited to the compatibility method. When you set up a node, you must ensure that the user ID of each of the other nodes is authorized to access that node or remote link from the initiating home node, as explained in “Linux user authorization (PE for Linux only)” on page 54.

Cluster based security (PE for AIX only)

With Cluster Based Security, the system administrator needs to ensure that UNIX Host Based authentication is enabled and properly configured on all nodes. Refer to the *IBM Parallel Environment: Installation* and the *IBM RSCT: Technical Reference* for what this entails.

From a user’s point of view, users will be required to have the proper entries in the **/etc/hosts.equiv** or **.rhosts** files, in order to ensure proper access to each node, as described in “AIX user authorization (PE for AIX only).”

AIX user authorization (PE for AIX only)

With AIX-based authentication, you are required to have an *.rhosts* file set up in your home directory on each of the remote processor nodes. Alternatively, your user id on the home node can be authorized in the */etc/host.equiv* file on each remote node. For information on the TCP/IP *.rhosts* file format, see *IBM AIX Version 5 Files Reference*.

Linux user authorization (PE for Linux only)

POE requires users to have remote execution authority in the system. Users must be authorized to system nodes via the **/etc/hosts.equiv** or **.rhosts** entries, as described in *IBM Parallel Environment: Installation*.

You are required to set up an **.rhosts** file in the home directory on each of the remote processor nodes. Alternatively, your user ID on the home node can be authorized in the **/etc/hosts.equiv** file on each remote node.

Checkpointing and restarting programs (PE for AIX only)

POE provides enhanced capabilities to checkpoint and later restart the entire set of programs that make up a parallel application, including the checkpoint and restart of POE itself. A number of previous restrictions for checkpointing have been removed as well.

Checkpointing programs (PE for AIX only)

Checkpointing is a method of periodically saving the state of job so that, if for some reason the job does not complete, it can be restarted from the saved state. At checkpoint time, checkpoint files are created on the executing machines. The checkpoint file of POE contains all information required to restart the job from the checkpoint files of the parallel applications.

Earlier versions of Parallel Environment's checkpoint/restart capability were based on user level checkpointing, with significant limitations. You can now checkpoint both batch and interactive jobs using LoadLeveler or PE in a system-initiated mode (external to the task) or in a user-initiated mode (internal to the task).

With system-initiated checkpointing, you can use the PE **poeckpt** command to checkpoint a non-LoadLeveler POE job. The applications are checkpointed at the point in their processing they happen to be when the checkpoint is issued. Checkpoint files are written for each task of the parallel application and for the POE executable itself. The locations of these files are controlled by the setting of the **MP_CKPTFILE** and **MP_CKPTDIR** environment variables. LoadLeveler also provides the **llckpt** command for checkpointing jobs being run under LoadLeveler (for more information, see *Tivoli Workload Scheduler LoadLeveler: Using and Administering*).

For a user-initiated checkpointing, the application may specify whether all tasks must issue the checkpoint request before the checkpoint occurs, or that one task of the application may cause the checkpoint of all tasks (and POE) to occur. The former is called a complete user-initiated checkpoint, and the latter is called a partial user-initiated checkpoint. In a complete user-initiated checkpoint, each task executes the application up to the point of the **mpc_init_ckpt** function call. In a partial user-initiated checkpoint, only one task executes the application up to the point of the **mpc_init_ckpt** call, and the remaining tasks are checkpointed at whatever point in their processing they happen to be when the checkpoint occurs, as in a system-initiated checkpoint.

In either system-initiated or user-initiated mode, **mpc_set_ckpt_callbacks** and **mpc_unset_ckpt_callbacks** calls can be made from within your parallel program. The *IBM Parallel Environment: MPI Programming Guide.* contains the specific information on these functions.

Using the settings of the **MP_CKPTDIR** and **MP_CKPTFILE** POE environment variables, the checkpoint data files are saved during the checkpointing phase, and the job is restarted by reading data from the checkpoint files during the restart phase. The **MP_CHECKDIR** and **MP_CHECKFILE** environment variables from previous releases are no longer used by POE.

When a checkpoint is taken, a set of checkpoint files is generated which consists of a POE checkpoint file and checkpoint files from each task of the parallel application. Each parallel task is checkpointed separately, and any processes created by a parallel task make up a checkpoint/restart group. The task checkpoint file contains information for all processes in the checkpoint/restart group. The checkpoint directory name is derived from the **MP_CKPTFILE** value (if it contains a full path name), the **MP_CKPTDIR** value, or the initial working directory. Tasks that change directories internally will not impact the place where the checkpoint file is written.

Note: When running a parallel program under LoadLeveler, the **MP_CKPTDIR** and **MP_CKPTFILE** environment variables are set by LoadLeveler. If the value for the checkpoint file name or directory is specified in the job command file, those values will override the current settings.

When the checkpointing files are created, tags are added to the names to differentiate between earlier versions of the files.

Restarting programs (PE for AIX only)

The PE **poerestart** command can be used to restart any interactive checkpointed jobs. POE is restarted first and it uses the saved information from its checkpoint file to identify the task checkpoint files to also restart. You can restart the application on the same set or different set of nodes, but the number of tasks and the task geometry must remain the same. When the restart function restarts a program, it retrieves the program state and data information from the checkpoint file. Note also that the restart function restores file pointers to the points at which the checkpoint occurred, but it does not restore the file content.

Checkpointing limitations (PE for AIX only)

When checkpointing a program, there are a few limitations of which you should be aware. You can find a complete list of the limitations in the *IBM Parallel Environment: MPI Programming Guide*. For example, when POE is invoked, the **CHECKPOINT** environment variable must be set to *yes* for POE and any of the parallel tasks to be checkpointable. LAPI programs can also be checkpointed if they meet the limitations.

Managing checkpoint files (PE for AIX only)

The ability to checkpoint or restart programs is controlled by the definition and availability of the checkpoint files, as specified by the **MP_CKPTFILE** environment variable.

The checkpoint files may be defined on the local file system (JFS) of the node on which the instance of the program is running, or they may be defined in a shared file system (such as NFS, AFS, DFS™, GPFS, etc.). When the files are in a local file system, then in order to perform process migration, the checkpoint files will have to be moved to the new system on which the process is to be restarted. If the old system crashed and is unavailable, it may not be possible to restart the program. It may be necessary, therefore, to use some kind of file management to avoid such a problem. If migration is not desired, it is sufficient to place checkpoint files in the local JFS file system.

The program checkpoint files can be large, and numerous. There is the potential need for significant amounts of available disk space to maintain the files. If possible, you should avoid using NFS, AFS, or DFS to manage checkpoint files. The nature of these systems is such that it takes a very long time to write and read large files. Instead, use GPFS or JFS.

If a local JFS file system is used, the checkpoint file must be written to each remote task's local file system during checkpointing. Consequently, during a restart, each remote task's local file system must be able to access the checkpoint file, from the previously checkpointed program, from the directory where the checkpoint file was written when the checkpoint occurred. This is of special concern when opting to restart a program on a different set of nodes from which it was checkpointed. The local checkpoint file may need to be relocated to any new nodes. For these reasons, it is suggested that GPFS be the file system best suited for checkpoint and restart file management.

A checkpoint/restart scenario (PE for AIX only)

A user's parallel application has been running on two nodes for six hours when the user is informed that the nodes must be taken down for service in an hour. The user expects the application to run for three more hours, and does not want to have to restart the application from the beginning on different nodes. The user set the **CHECKPOINT** environment variable to *yes* before issuing the POE command, so that the operating system would allow the checkpoint to occur. Furthermore, the user set the **MP_CKPTDIR** environment variable to a GPFS directory, **/gpfs**, so that the checkpoint files would be accessible from other nodes. The user also set the **MP_CKPTFILE** environment variable to the name of the application, *9hourjob*, so it can be easily identified later.

After setting the **MP_CKPTDIR** and **MP_CKPTFILE** environment variables, the user obtains the process identifier of the POE process. Then, the user issues the **poeckpt** command, along with the **-k** option so that the tasks will be terminated once the checkpoints are successfully completed. The checkpoints of the parallel tasks are taken first, and then the checkpoint of POE occurs. The **poeckpt** command reports the following:

```
poeckpt: Checkpoint of POE process 12345 has succeeded.
poeckpt: The /gpfs/9hourjob.0 checkpoint file has been created.
```

The filename indicated in the output, **/gpfs/9hourjob**, is the checkpoint file of the POE process which will be used later when the parallel application is restarted. The *.0* suffix is a tag used to allow one set of previously successful checkpoint files to be saved (a subsequent checkpoint on this program, although unlikely in this scenario, would use tag 1).

To determine the behavior of the checkpoint function, the user issues:

```
ls /gpfs/9hour*
```

and sees the following output:

```
/gpfs/9hourjob.0    /gpfs/9hourjob.0.0   /gpfs/9hourjob.1.0
```

The additional files besides the one reported by the output are the checkpoint files from each of the tasks that made up the parallel application. The last *0* in the task checkpoint files represents the checkpoint tag as described previously. The digit before the tag is the task number within the parallel application.

The user finds two other nodes that can be used to restart the parallel job and sets up a host.list, containing these two host names, in the directory from which the user will run the **poerestart** command. The user issues:

```
poerestart /gpfs/9hourjob.0
```

The restarted POE from this checkpoint file *remembers* the names of the task checkpoint files to restart from, tells the Partition Manager Daemon on each node to restart each parallel task from their respective checkpoint file, and the parallel application is running again. The job completes in three hours, and produces the same results as it would have had it run for nine hours on the original nodes.

Managing task affinity on large SMP nodes

Large SMP nodes are organized around components called Multi-chip Modules (MCM). An MCM contains several processors, I/O buses, and memory. While a processor in an MCM can access the I/O bus and memory in another MCM, demanding applications may see improved performance if the processor, the memory it uses, and the I/O adapter it connects to, are all in the same MCM.

PE provides the environment variable **MP_TASK_AFFINITY** to control the placement of the tasks of a parallel job so that the task will not be migrated between MCM's during its execution.

Note: For AIX V5.3 TL 5300-05, it is recommended that the system administrator configure the computing node to use memory affinity with various combinations of the **memplace_*** options of the **vmo** command.

PE allows you to manage task affinity for two different types of jobs:

- Stand-alone POE jobs that do not interact with LoadLeveler to establish affinity

  In this case, POE uses the values provided with the **MP_TASK_AFFINITY** environment variable and the **pm_set_affinity** routine to manage affinity. In the stand-alone scenario, either LoadLeveler is not installed, or it is installed but has not been configured for affinity.
- Interactive POE jobs that work in conjunction with LoadLeveler to establish affinity

  In this case, POE uses the values that are provided with the **MP_TASK_AFFINITY** environment variable and converts them to the corresponding LoadLeveler JCF keyword values, and then passes them to LoadLeveler.

  In this scenario, POE relies on LoadLeveler to handle scheduling affinity, based on the LoadLeveler job control file keywords that POE sets up when the job is submitted. Memory and task affinity must be enabled in the LoadLeveler configuration file (using the **RSET_SUPPORT** keyword). Also, for jobs that are run with processor or core affinity, the LoadLeveler configuration and administration files must be configured as shown below:

  For the LoadLeveler configuration file, consumable CPUs must be configured. For example:

  ```
  SCHEDULE_BY_RESOURCES = ConsumableCpus
  rset_support = rset_msm_affinity
  ```

For the LoadLeveler administration file, **ConsumableCpus(all)** should be defined for the **resources** keyword in the machine stanzas. One way of doing this is to specify **resources = ConsumableCpus(all)** in the default machine stanza, which would be inherited by all machines. For example:

```
default: type = machine
resources = ConsumableCpus(all)
```

For more information about LoadLeveler configuration, refer to *IBM Tivoli Workload Scheduler LoadLeveler: Using and Administering*.

Note: In order to use LoadLeveler in conjunction with POE for scheduling affinity, LoadLeveler 3.3.1 or later (for AIX) or LoadLeveler Version 3.4.3 (for Linux) must be installed.

POE specifies the affinity options as LoadLeveler preferences, not requirements, meaning that if the affinity option cannot be satisfied, the job may still run. If you want a different set of LoadLeveler scheduling affinity options, you must use your own LoadLeveler JCF file, and not specify POE's **MP_TASK_AFFINITY** option, which will result in POE setting up the LoadLeveler JCF options as described above.

PE, in conjunction with LoadLeveler, provides affinity support for OpenMP applications. For more information, see "OpenMP task affinity support" on page 61.

Parallel Environment provides the environment variable **MP_TASK_AFFINITY** to control the placement of tasks in a parallel job so that the task will not be migrated between MCMs during its execution. The possible **MP_TASK_AFFINITY** values are as follows:

- **MCM –** Specifies that the tasks are allocated in a round-robin fashion among the MCMs attached to the job by WLM. By default, the tasks are allocated to all the MCMs in the node.

  When run under LoadLeveler, POE sets the LoadLeveler **MCM_AFFINITY_OPTIONS** and **RSET** keywords to allow LoadLeveler to handle scheduling affinity, as follows:
  - Sets the **MCM_AFFINITY_OPTIONS** keyword to **MCM_MEM_PREF**, **MCM_SNI_NONE**, and **MCM_DISTRIBUTE**
  - Sets the **RSET** keyword to **RSET_MCM_AFFINITY**.
- **SNI –** Specifies that the tasks are allocated to the MCM in common, with the first adapter assigned to the task by LoadLeveler. This value applies only to User Space MPI jobs. **SNI** should not be specified for IP jobs.

  When run under LoadLeveler, POE sets the LoadLeveler **MCM_AFFINITY_OPTIONS** and **RSET** keywords to allow LoadLeveler to handle scheduling affinity, as follows:
  - Sets the **MCM_AFFINITY_OPTIONS** keyword to **MCM_SNI_PREF** and **MCM_DISTRIBUTE**
  - Sets the **RSET** keyword to **RSET_MCM_AFFINITY**.

  When using an InfiniBand adapter for stand-alone POE jobs (run without LoadLeveler), only InfiniBand adapters with valid *ibm,associativity* vectors are supported. Also note that with Linux, only InfiniBand adapters are supported.
- **CORE –** Specifies that each MPI task runs on a single physical processor core. If simultaneous multithreading (SMT) is disabled, this may be one CPU. If SMT is enabled, it may be two CPUs.

When an interactive POE job is run under LoadLeveler, POE sets the LoadLeveler **RSET** and **task_affinity** keywords to allow LoadLeveler to handle scheduling affinity, as follows:

  – Sets the **RSET** keyword to **RSET_MCM_AFFINITY**
  – Sets the **MCM_AFFINITY_OPTIONS** keyword to **MCM_MEM_PREF**, **MCM_SNI_NONE**, and **MCM_ACCUMULATE**.
  – Sets the **task_affinity** keyword to `@task_affinity=core(1)`.

- **CPU** – Specifies that each MPI task runs on a single logical CPU.

  When an interactive POE job is run under LoadLeveler, POE sets the LoadLeveler **RSET** and **task_affinity** keywords to allow LoadLeveler to handle scheduling affinity, as follows:

  – Sets the **RSET** keyword to **RSET_MCM_AFFINITY**
  – Sets the **MCM_AFFINITY_OPTIONS** keyword to **MCM_MEM_PREF**, **MCM_SNI_NONE**, and **MCM_ACCUMULATE**.
  – Sets the **task_affinity** keyword to `@task_affinity=cpu(1)`.

- **CORE:*n*** – Specifies the number of processor cores to which the threads of an MPI task are constrained (one thread per core), where ***n*** is an integer between 1 and 99999. When you specify the value for ***n*,** you must precede it with a colon (**:*n***).

  Specifying **CORE:*n*** signifies the use of OpenMP support, which requires XL Fortran Version 11 PTF1 (or later) or XL C/C++ Version 9.0 (or later). This option applies only to OpenMP jobs that are run interactively with LoadLeveler, using the AIX operating system, and requires LoadLeveler 3.4.3, or later (when an interactive POE job is run under LoadLeveler). For more information, see "OpenMP task affinity support" on page 61.

  When an interactive POE job is run under LoadLeveler, POE sets the LoadLeveler **RSET**, **task_affinity**, and **parallel_threads** keywords to allow LoadLeveler to handle scheduling affinity, as follows:

  – Sets the **RSET** keyword to **RSET_MCM_AFFINITY**
  – Sets the **task_affinity** keyword to `@task_affinity=core(n)`
  – Sets the **parallel_threads** keyword to `@parallel_threads=n`.
  – Sets the **OMP_NUM_THREADS** and **XLSMPOPTS** environment variables, as appropriate. See "OpenMP task affinity support" on page 61 for more information.

- **CPU:*n*** – Specifies the number of logical CPUs to which the threads of an MPI task are constrained (one thread per CPU), where ***n*** is an integer between 1 and 99999. When you specify the value for ***n*,** you must precede it with a colon (**:*n***).

  Specifying **CPU:*n*** signifies the use of OpenMP support, which requires XL Fortran Version 11 PTF1 (or later) or XL C/C++ Version 9.0 (or later). This option applies only to OpenMP jobs that are run interactively with LoadLeveler, using the AIX operating system, and requires LoadLeveler 3.4.3, or later (when an interactive POE job is run under LoadLeveler). For more information, see "OpenMP task affinity support" on page 61.

  When an interactive POE job is run under LoadLeveler, POE sets the LoadLeveler **RSET**, **task_affinity**, and **parallel_threads** keywords to allow LoadLeveler to handle scheduling affinity, as follows:

  – Sets the **RSET** keyword to **RSET_MCM_AFFINITY**
  – Sets the **task_affinity** keyword to `@task_affinity=cpu(n)`
  – Sets the **parallel_threads** keyword to `@parallel_threads=n`.

   – Sets the **OMP_NUM_THREADS** and **XLSMPOPTS** environment variables, as appropriate. See "OpenMP task affinity support" on page 61 for more information.
- **mcm-list** – Specifies that the tasks are assigned on a round-robin basis to this set, within the constraint of an inherited rset, if any. **mcm-list** specifies a set of system level (LPAR) logical MCMs to which tasks can be attached. Assignments to MCMs that are outside the constraint set are attempted, but will fail. If a single MCM number is specified as the list, all tasks are assigned to that MCM. This option is only valid when running without LoadLeveler, or with LoadLeveler Version 3.2 (or earlier), which does not support scheduling affinity.
- **-1** – Specifies that no affinity request will be made (disables task affinity).

Note that the **MP_TASK_AFFINITY** settings cannot be used for batch POE jobs that are run with LoadLeveler (if values are specified, they are ignored). Instead, if a batch job requires memory affinity, the LoadLeveler **RSET** and **MCM_AFFINITY_OPTIONS** keywords need to be specified in the LoadLeveler job command file. Refer to *IBM Tivoli Workload Scheduler LoadLeveler: Using and Administering* for more information.

IMPORTANT NOTE:

The stand-alone POE task affinity function relies on the pm_set_affinity program, which is a set-user-on-execution (setuid) binary file. The pm_set_affinity program is shipped in **/usr/lpp/ppe.poe/bin** (for AIX) or **/opt/ibmhpc/ppe.poe/bin** (for Linux), and is owned by the root system user (-r-sr-xr-x root:system | root:root). pm_set_affinity allows system administrators to temporarily grant non-root users the ability to modify the priority of the jobs that are executed by POE. However, it is important to note that disabling the set bit (s bit) on this program prevents general users from being able to modify the CPU or adapter affinity of jobs. Note that this limitation only applies if you have invoked the POE task affinity function using the **MP_TASK_AFFINITY** environment variable or the **-task_affinity** command line flag, ***and*** POE is running ***without*** LoadLeveler.

If you invoke the task affinity function with **MP_TASK_AFFINITY** or **-task_affinity**, you will only see output that indicates an affinity error occurred if you set the **-infolevel** command line flag to at least 4, and also set **MP_PMDLOG** to **yes**. So, in this case, when running without LoadLeveler, your output would look similar to this:

```
-----------------------------------------------
D1: <L4>: pm_set_affinity failed, MP_PHYSICAL_MCM value: -1

example pmdlog output:
pm_set_affinity rc = 0xffffffff
-----------------------------------------------
```

The adverse effects on the POE task affinity function of running the AIX **fpm** command are limited only to the user's ability to set MCM or adapter affinity; otherwise, the job should continue to run.

Smaller SMP nodes may be organized around Dual Chip Modules (DCMs). From POE's viewpoint, a DCM is equivalent to an MCM, and **MP_TASK_AFFINITY=MCM** will round-robin tasks among DCMs. Multithreaded applications may need to be aware that a DCM has only 1 or 2 processor cores, while MCMs have up to 8 processor cores.

The **rset_query** (for AIX) or **cpuset_query** (for Linux) commands can also be used to verify that memory affinity assignments are being performed correctly.

The output of the **rset_query** and **cpuset_query** commands shows the number of available processors, memory pools, memory, and processors in resource sets, on a per-MCM or per-DCM basis. These commands take no options or parameters. Each of these commands needs to be invoked under POE as a parallel job so that it displays the MCM assignments POE is using when running. It is also possible to invoke either of these commands as part of a multiple step POE job, where **rset_query** or **cpuset_query** is run as the first step prior to running the application code in a subsequent step. Using **MP_LABELIO=yes** and **MP_STDOUTMODE=ordered** may help you interpret the output more easily.

OpenMP task affinity support

PE, in conjunction with LoadLeveler, provides affinity support for OpenMP applications. LoadLeveler provides the **parallel_threads** keyword to handle requests made, through the OpenMP **XLSMPOPTS** environment variable, for the binding of individual threads. POE copies the user's environment and uses the values set for the **OMP_NUM_THREADS** environment variable or the **parthds** suboption of **XLSMPOPTS** to determine the setting of the LoadLeveler **parallel_threads** JCF keyword. Note that **OMP_NUM_THREADS** takes precedence over **XLSMPOPTS parthds**.

With interactive POE jobs that use LoadLeveler for managing affinity, POE checks to see if values were provided for the **MP_TASK_AFFINITY CORE:n** or **CPU:n** options and sets the same value for the **parallel_threads** keyword. This same value also needs to be provided with the **OMP_NUM_THREADS** or **XLSMPOPTS parthds** variables. Because of the precedence that exists between these values, POE handles them as follows:

- If **OMP_NUM_THREADS** is not set, POE checks **XLSMPOPTS parthds**. If neither have been set, POE checks the **MP_TASK_AFFINITY CORE:n** and **CPU:n** values. If **CORE:n** or **CPU:n** have been set, POE uses that value for **OMP_NUM_THREADS** and the LoadLeveler **parallel_threads** keyword.
- If **OMP_NUM_THREADS** is set, POE compares that value to the value specified by **MP_TASK_AFFINITY CORE:n** or **CPU:n**. If the two values are different, POE issues an informational message and uses the value specified for **OMP_NUM_THREADS** as the value for the LoadLeveler **parallel_threads** keyword (**MP_TASK_AFFINITY CORE:n** and **CPU:n** are ignored).
- If neither the **OMP_NUM_THREADS** nor **XLSMPOPTS parthds** variables have been set, POE uses the setting for **MP_TASK_AFFINITY CORE:n** or **CPU:n** for both **OMP_NUM_THREADS** and **XLSMPOPTS parthds**. Note that this may result in poor performance because it may lead to oversubscription of processors when the task count is greater than one on any node.

Note: If the number of threads will be changed over the course of an OpenMP program's execution, the use of **MP_TASK_AFFINITY** is not recommended. In this case, you should perform your own thread binding within the application.

With batch POE jobs, LoadLeveler uses the value specified for **OMP_NUM_THREADS** as the value for **parallel_threads**. The **MP_TASK_AFFINITY** environment variable settings are ignored for batch requests.

For OpenMP support, the XL Fortran V11 runtime library, at the PTF1 level, or XL C/C++ Version 9.0 (or later) is required.

Running POE from a shell script (PE for AIX only)

Due to an AIX limitation, if the program being run by POE is a shell script AND there are more than 5 tasks being run per node, then the script must be run under ksh93 by using:

```
#!/bin/ksh93
```

on the first line of the script.

Using POE with MALLOCDEBUG (PE for AIX only)

Submitting a POE job that uses **MALLOCDEBUG** with an *align:n* option of other than 8 may result in undefined behavior. To allow a POE parallel program to run with an *align:n* option other than 8, you will need to create a script file.

For example, say the POE program is named *myprog*. You could create the following script file:

```
MALLOCTYPE=debug
MALLOCDEBUG=align:0
myprog myprog_options
```

Once you had created the script file, you could then run the script file using the **poe** command. For example, if the script file were named *myprog.sh* you would enter:

```
poe myprog.sh <poe_options> <myprog_options>
```

Instead of:

```
poe myprog <poe_options> <myprog_options>
```

Using POE with AIX large pages (PE for AIX only)

Memory requests in applications that use large pages in *mandatory* mode may fail unless there are a minimum of 16 large pages (16M each) available for each parallel task that makes a memory request. If any task requests >256M, an additional 16 large pages must be available for each task, for each additional 256M requested. In addition, unless the following workaround is used, an additional 16 large pages must be available for the POE process as well.

To avoid having the POE process use mandatory large pages, do not set the **LDR_CNTRL** environment variable to **LARGE_PAGE_DATA=M** before invoking POE. The value of this environment variable, (M in this case) is case sensitive. Instead, use POE to invoke a script that first exports the environment variable, and then invokes the parallel program.

POE provides the **MP_TLP_REQUIRED** environment variable and **-tlp_required** command line flag to ensure that running jobs have been compiled with large pages. The options are:

warn	POE issues a warning message for any job that was not compiled with large pages, and the job will continue to run.
kill	POE detects and kills any job that was not compiled with large pages.
none	POE takes no action (this is the default).

Using **MP_TLP_REQUIRED** may help avoid system failures due to a lack of paging space, where large memory applications are executed without being compiled to use large pages.

Chapter 3. Managing POE jobs

There are a number of tasks you need to understand that are related to managing POE jobs. These tasks include how to allocate nodes, saving core files, improving parallel job performance, stopping and cancelling a job, detecting remote node failures, and so on.

Multi-task core file

With the **MP_COREDIR** environment variable, you can create a separate directory to save a core file for each task. The corresponding command line option is **-coredir**. Creating this type of directory is useful when you are running a parallel job on one node (AIX) or using a shared file system (Linux), and your job dumps a core file.

By checking this directory, you can see which task dumped the file. When setting **MP_COREDIR**, you specify the first attribute of the directory name. The second attribute is the task id. If you do not specify a directory, the default is *coredir*. The subdirectory containing each task's core file is named *coredir.taskid*.

You can also disable the creation of a new subdirectory to save a core file, by specifying **-coredir** or **MP_COREDIR** with a value of *none*. When disabled, core files will be written to **/tmp** instead of your current directory.

Disabling the creation of a new subdirectory may be necessary in situations where programs are abnormally terminating due to memory allocation failures, (for example, a malloc() call is the result of the original core file). In these cases, setting **-coredir** or **MP_COREDIR** to *none* may prevent a situation where POE could hang as a result of a memory allocation problem while it is attempting to create a new subdirectory to hold the core file.

The following examples show what happens when you set the environment variable.

PE for AIX examples:

```
Example 1:


MP_COREDIR=my_parallel_cores

MP_PROCS=2


run generates corefiles


Corefiles will be located at:


/current directory/my_parallel_cores.0/core

/current directory/my_parallel_cores.1/core
```

Example 2:

```
MP_COREDIR not specified

MP_PROCS=2

run generates corefiles

Corefiles will be located at:

/current directory/coredir.0/core

/current directory/coredir.1/core
```

Example 3:

```
MP_COREDIR=none

MP_PROCS=2

run generates corefiles

Corefiles will be located at:

/tmp/core
```

PE for Linux examples:

Example 1:

```
MP_COREDIR=my_parallel_cores

MP_PROCS=2

run generates corefiles

Corefiles will be located at:

/current directory/my_parallel_cores.0/core.23456.0

/current directory/my_parallel_cores.1/core.23457.1
```

Example 2:

```
MP_COREDIR not specified
```

```
MP_PROCS=2


run generates corefiles


Corefiles will be located at:


/current directory/coredir.0/core.23456.0

/current directory/coredir.1/core.23457.1


Example 3:


MP_COREDIR=none

MP_PROCS=2


run generates corefiles


Corefiles will be located at:


/tmp/core.23457.1
```

Note: If the tasks that you run produce the same process or task numbers as previous tasks, only the last core file, with that process ID and task ID combination, are saved. Previous files may be overwritten.

Support for performance improvements

Parallel job performance can be greatly affected by the AIX or Linux operating system's network settings and by the values that you assign to a set of environment variables that are recognized by POE and by various libraries of the protocol stack.

See *IBM Parallel Environment: Installation* for information on how to tune the AIX or Linux operating system and network devices for better parallel job performance.

See the *IBM Parallel Environment: MPI Programming Guide* for information on how various environment variables affect the performance of a parallel job.

The following sections discuss how to use the **MP_BUFFER_MEM** and **MP_CSS_INTERRUPT** environment variables.

Using MP_BUFFER_MEM

The **MP_BUFFER_MEM** environment variable specifies the size of the Early Arrival (EA) buffer that is used by the communication subsystem to buffer eagerly sent **point-to-point** messages that arrive before there is a matching receive posted. This value can also be specified with the **-buffer_mem** command line flag. The command line flag overrides a value set with the environment variable.

A separate Early Arrival buffer exists for eagerly sent collective communications messages. The size of this buffer can be specified with the **MP_CC_BUF_MEM** environment variable. For more information, see "MP_CC_BUF_MEM details" on page 240.

The total amount of point-to-point Early Arrival buffer space allocated by a task is controlled by **MP_BUFFER_MEM**. If a single value is given, it is important for good performance that the amount of memory specified by **MP_BUFFER_MEM** be sufficient to hold a reasonable number of unmatched messages of size up to the **eager_limit** from every possible sender. If necessary, PE may reduce the **eager_limit** to achieve this. The memory is preallocated and preformatted for efficiency. Default values are usually sufficient for jobs up to 512 tasks.

If two values (M1,M2) are given for **MP_BUFFER_MEM**, the first value specifies the amount of preformatted memory (and presumably is an estimate of the actual memory requirement for Early Arrival messages); the second value is used as the maximum requirement for Early Arrival buffering. PE ensures that this memory requirement is not exceeded under any circumstances by limiting the number of outstanding **eager_limit** messages from any sender.

This environment variable has two forms, as follows:

```
MP_BUFFER_MEM=pre_allocated_size

MP_BUFFER_MEM=pre_allocated_size,maximum_size
```

With AIX, the first form is compatible with prior releases and is still suitable for most applications. The second provides flexibility that may be useful for some applications, in particular at large task counts.

Examples:

```
export  MP_BUFFER_MEM=32M
export  MP_BUFFER_MEM=32M,128M
export  MP_BUFFER_MEM=0,128M
export  MP_BUFFER_MEM=,128M
```

The **pre_allocated_size** argument is used to specify the size of the buffer to be preallocated and reserved for use by the MPI library. This space is allocated during initialization. If you omit this argument, or if you do not specify the **MP_BUFFER_MEM** variable at all, the MPI library assigns a default value of 64 MB for both User Space and IP applications. The maximum allowable value is 256 MB.

For the **pre_allocated_size** argument, you may specify a positive number or zero, or provide the comma but omit the value. If the positive number is greater than the minimum size that is needed by MPI for correct operation and no greater than 256MB, a buffer of this size will be preallocated. An omitted value tells the Parallel Environment implementation of MPI to use the default preallocated EA buffer size. A zero tells the Parallel Environment implementation of MPI to use the minimum workable EA preallocation. You must specify the value in bytes, and you may use K (kilobytes), M (megabytes), or G (gigabytes) as part of the specification.

The **maximum_size** argument is used to specify the maximum size to which the EA buffer can temporarily grow when the preallocated portion of the early arrival buffer has been filled. If the behavior of your application is such that the extra space really must be used, it will be borrowed from the heap as needed. In that case, it can be regarded as an ongoing contention for memory between the MPI library and the application. Therefore, if your application actually uses more than the preallocated space, you should consider raising the preallocation to cover it. That is, if you can afford to have the extra memory used for early arrivals, then it probably makes sense to preallocate it. If you cannot spare the extra memory, it may be better to remove the **maximum_size** value and let MPI constrain eager messages to stay within the memory you can afford to preallocate. See the description of **MP_STATISTICS** in "poe" on page 183.

You may specify a positive number or omit the comma and specification. You must specify the value in bytes, and you may use K (kilobytes), M (megabytes), or G (gigabytes) as part of the specification. Note also that for 64-bit applications, the maximum buffer size may exceed 4 gigabytes.

Important: You can use the **-buffer_mem** command line flag to specify the **pre_allocated_size** and **maximum_size** values or **pre_allocated_size** alone. However, note that the two values you specify must be separated by a comma, and blanks are not allowed unless you surround the values with quotes. The following examples show correct use of the **-buffer_mem**:

```
poe -buffer_mem 32M

poe -buffer_mem 32M,64M

poe -buffer_mem '32M, 64M'

poe -buffer_mem ,64M
```

To preallocate the entire EA buffer, specify **MP_BUFFER_MEM** and provide a single value. The value you provide will be assigned to both the **pre_allocated_size** and **maximum_size** arguments. The maximum allowable value is 256 MB.

The default value for **MP_BUFFER_MEM** is 64 MB for both User Space and IP applications.

If you are using PE for AIX and you will be checkpointing a program, be aware that the amount of space needed for the checkpoint files will include the entire preallocated buffer, even if only parts of it are in use. The extent to which the heap has been allocated also affects the size of the checkpoint files.

Important: Setting the **MP_BUFFER_MEM** maximum to a value greater than the preallocated size implies that you are either able to commit enough heap memory to early arrivals to cover the difference, or that you are confident that the maximum demand will not occur and you have sufficient memory for the actual peak. If the malloc() fails due to unexpected peaks in EA buffer demand and insufficient memory in the system, the job is terminated. For most well-structured MPI applications, you will see only modest demand for early arrival space, even when you set a high upper bound.

Note that the MPI library adds 64K to all of the values you specify, which it uses for internal management of the Early Arrival buffer.

Improving performance with MP_CSS_INTERRUPT

The **MP_CSS_INTERRUPT** environment variable may take the value of either **yes** or **no**. By default it is set to **no**. In certain applications, setting this value to **yes** will provide improved performance.

The following briefly summarizes some general application characteristics that could potentially benefit from setting **MP_CSS_INTERRUPT=yes**.

Applications that have the following characteristics may see performance improvements from setting the POE environment variable **MP_CSS_INTERRUPT** to **yes**:

- Applications that have non-synchronized sets of send or receive pairs. In other words, the send from node0 is issued at a different point in time with respect to the matching receive in node1.
- Applications that do not issue waits for nonblocking send or receive operations immediately after the send or receive, but rather do some computation prior to issuing the waits.

In all of the previous cases, the application is taking advantage of the asynchronous nature of the nonblocking communication subroutines. This essentially means that the calls to the nonblocking send or receive routines do not actually ensure the transmission of data from one node to the next, but only post the send or receive and then return immediately back to the user application for continued processing. However, since the User Space protocol executes within the user's process, it must regain control from the application to advance asynchronous requests for communication.

The communication subsystem can regain control from the application in any one of three different methods:

1. Any subsequent calls to the communication subsystem to post send or receive, or to wait on messages.
2. A timer pop occurring periodically to allow the communication subsystem to do recovery for transmission errors and to make progress on pending nonblocking communications.
3. If the value of **MP_CSS_INTERRUPT** is set to **yes**, the communication subsystem device driver will notify the user application when data is received or buffer space is available to transmit data.

Method 1 and Method 2 are always enabled. Method 3 is controlled by the POE environment variable **MP_CSS_INTERRUPT**, and is enabled when this variable is set to **yes**.

For applications that post nonblocking sends or receives, and turn to computation for a period before posting the wait, any communication that is to happen while the application is computing must occur through the second or third of these three methods. If **MP_CSS_INTERRUPT** is not enabled, only the timer pop method is available to advance communication and time pops are far enough apart so they make very slow progress. The goal in overlapping communication and computation is to hide latency by doing useful computation while the data moves. In the ideal case, the data will have been transferred by the time the computation finishes, and the deferred wait can return immediately.

For example, consider the following application template, where two tasks execute the same code:

```
 LOOP
  MPI_ISEND (A, ..,partner,.., send_req)
  MPI_IRECV (B, ..,partner,.., recv_req)
  MPI_WAIT (recv_req, ......)
  COMPUTE LOOP1  /* uses data in B */
  MPI_WAIT (send_req, ......)
  COMPUTE LOOP2 (modifies A)
END LOOP
```

In this example, data B is guaranteed to be received by the return from the wait for recv_req and it is likely the return from the wait call will be delayed while the data is actually flowing in. Data B can then be safely used in the COMPUTE LOOP1. Data A is not guaranteed to be fully sent until the wait for send_req returns, but this is acceptable for the task in COMPUTE LOOP1 because it can compute with data B.

In this simple example, it is likely that one task will receive data B during the wait for recv_req and enter COMPUTE LOOP1 before the send of data A has finished. When this happens, the rest of the work to send data A will need to progress while the task is computing. This is important for two reasons:

- A task that finishes its receive and goes on to COMPUTE LOOP1 before also finishing the send will stall its partner in its receive while waiting for that send to finish. The stalling of the task in its receive is directly related to the noncontinuous flow of communication from the task that turned to computing. With **MP_CSS_INTERRUPT=yes**, each time the communication is ready to make more progress on the send, the communication subsystem device driver interrupts the computation just long enough to advance the communication. Therefore, data flow from the task that is computing to the partner that is stalled is maintained and that stalled task also gets to move on to computation.
- By the time COMPUTE LOOP1 is done, it is likely that data A has all been sent and the return from the wait can be prompt.

The reason this example is simple is that it involves a race condition that makes it likely one task will move on to computation while the other is still waiting for a communication that the computing task is no longer concerned with. **MP_CSS_INTERRUPT** makes sure that communication makes reasonable progress but it will be slower than if the send had also been waited. Because the outer loop makes both tasks move in lock step, any delay that the race-winning task causes its partner, by leaving it stuck in a receive wait while the partner computes, will later delay that winner when it needs to postpone its next iteration until the delayed task catches up.

Stopping a POE job

You can stop (suspend) an interactive POE job by pressing <**Ctrl-z**> or by sending POE a SIGTSTP signal. POE stops, and sends a SIGSTOP signal to all the remote tasks, which stops them.

To resume the parallel job, issue the **fg** or **bg** command to POE. A SIGCONT signal will be sent to all the remote tasks to resume them.

Cancelling and killing a POE job

You can cancel a POE job by pressing <**Ctrl-c**> or <**Ctrl-**>. This sends POE a SIGINT or SIGQUIT signal respectively. POE terminates all the remote tasks and exits.

If POE on the home node is killed or terminated before the remote nodes are shut down, direct communication with the parallel job will be lost. In this situation, use the **poekill** script as a POE command, or individually via **rsh**, to terminate the partition. **poekill** kills all instantiations of the program name on a remote node by sending it a SIGTERM signal. See the description of the **poekill** command in Chapter 6, "Parallel Environment commands," on page 129, and the **poekill** script in **/usr/lpp/ppe.poe/bin** (for AIX) or **/opt/ibmhpc/ppe.poe/bin** (for Linux).

Note: *Do not* kill the **pmd**s using the **poekill** command. Doing so will prevent your remote processes from completing normally.

Detecting remote node failures

POE and the Partition Manager use a *pulse* detection mechanism to periodically check each remote node to ensure that it is actively communicating with the home node.

You specify the time interval (or *pulse* interval), of these checks with the **-pulse** flag or the **MP_PULSE** environment variable. During an execution of a POE job, POE and the Partition Manager daemons check at the interval you specify that each node is running. When a node failure is detected, POE terminates the job on all remaining nodes and issues an error message.

The default pulse interval is 600 seconds (10 minutes). You can increase or decrease this value with the **-pulse** flag or the **MP_PULSE** environment variable. To completely disable the pulse function, specify an interval value of 0 (zero). If you are using PE for AIX, note that for the PE debugging facility, **MP_PULSE** is disabled.

If you are using PE for Linux and you plan to debug parallel applications under POE, disable the POE pulse mechanism by setting the **MP_PULSE** environment variable or the **-pulse** command line flag to zero.

Submitting a batch POE job using TWS LoadLeveler

To submit a batch POE job using LoadLeveler, you need to build a LoadLeveler job file.

The LoadLeveler job file specifies:
- The number of nodes to be allocated
- Any POE options, passed via environment variables using LoadLeveler's *environment* keyword, or passed as command line options using LoadLeveler's *argument* keyword.
- The path to your POE executable (usually **/usr/bin/poe**).
- Adapter specifications using the network keyword.

The following POE environment variables, or associated command line options, are validated, but not used, for batch jobs submitted using LoadLeveler.

- **MP_ADAPTER_USE**
- **MP_CPU_USE**
- **MP_DEVTYPE**
- **MP_EUIDEVICE**
- **MP_EUILIB**
- **MP_HOSTFILE**
- **MP_INSTANCES**
- **MP_MSG_API** (except for programs that use LAPI and also use the LoadLeveler **requirements** keyword to specify **Adapter=**"*hps_user*")
- **MP_NODES**
- **MP_PROCS**
- **MP_RDMA_COUNT** (PE for AIX only)
- **MP_RESD**
- **MP_RETRY**
- **MP_RETRYCOUNT**
- **MP_RMPOOL**
- **MP_SAVEHOSTFILE**
- **MP_TASK_AFFINITY**
- **MP_TASKS_PER_NODE**
- **MP_USE_BULK_XFER**

For examples, see "PE for AIX example of submitting a batch POE job using TWS LoadLeveler" and "PE for Linux example of submitting a batch POE job using TWS LoadLeveler" on page 75.

PE for AIX example of submitting a batch POE job using TWS LoadLeveler

To run **myprog** on five nodes, using a Token ring adapter for IP message passing, with the message level set to the **info** threshold, you could use the following LoadLeveler job file. The arguments **myarg1** and **myarg2** are to be passed to **myprog**.

```
#!/bin/ksh

# @ input = myjob.in

# @ output = myjob.out

# @ error = myjob.error

# @ environment = COPY_ALL; \

    MP_INFO_LEVEL=2

# @ executable = /usr/bin/poe

# @ arguments = myprog myarg1 myarg2

# @ min_processors = 5
# @ requirements = (Adapter == "tokenring")

# @ job_type = parallel
```

To run **myprog** on 12 nodes from pool 2, using the User Space message passing interface with the message threshold set to **attention**, you could use the following LoadLeveler job file. See the documentation provided with the LoadLeveler program product for more information.

```
#!/bin/ksh

# @ input = myusjob.in

# @ output = myusjob.out

# @ error = myusjob.error

# @ environment = COPY_ALL;

# @ executable = /usr/bin/poe

# @ arguments = myprog -infolevel 1

# @ min_processors = 12

# @ requirements = (Pool == 2) && (Adapter == "hps_user")

# @ job_type = parallel

# @ checkpoint = no
```

Note:

1. The first token of the *arguments* string in the LoadLeveler job file must be the name of the program to be run under POE, unless:
   - You use the **MP_CMDFILE** environment variable or the **-cmdfile** command line option
   - The file you specify with the keyword *input* contains the name(s) of the programs to be run under POE.
2. When setting the environment string, make sure that no white space characters follow the backslash, and that there is a space between the semicolon and backslash.
3. When LoadLeveler allocates nodes for parallel execution, POE and task 0 will be executed on the same node.
4. When LoadLeveler detects a condition that should terminate the parallel job, a SIGTERM will be sent to POE. POE will then send the SIGTERM to each parallel task in the partition. If this signal is caught or ignored by a parallel task, LoadLeveler will ultimately terminate the task.
5. Programs that call the **usrinfo** function with the **getinfo** parameter, or programs that use the **getinfo** function, are not guaranteed to receive correct information about the owner of the current process.
6. Programs that use LAPI and also the LoadLeveler *requirements* keyword to specify **Adapter=**"**hps_user**", must set the **MP_MSG_API** environment variable or associated command line option accordingly.
7. If the value of the **MP_EUILIB**, **MP_EUIDEVICE**, or **MP_MSG_API** environment variable that is passed as an argument to POE differs from

the specification in the network statement of the job command file, the network specification will be used, and an attention message will be printed.

For more information, refer to *Tivoli Workload Scheduler LoadLeveler: Using and Administering*.

PE for Linux example of submitting a batch POE job using TWS LoadLeveler

To run an MPI program (**myprogram**) using an Ethernet adapter (**eth0**) for IP message passing, with the message level set to the **info** threshold, you could use the following LoadLeveler job command file. The arguments **myarg1** and **myarg2** are passed to **myprogram** and the output and error files are created with names that include the process ID.

```
#!/bin/ksh
#@ job_type = parallel
#@ environment = COPY_ALL; \
MP_INFOLEVEL=2
#@ node_usage = shared
#@ network.mpi = eth0,shared,ip
#@ class = Parallel
#@ node = 2
#@ tasks_per_node = 2
#@ queue
#@ executable = /usr/bin/poe
#@ arguments = myprogram myarg1 myarg2
#@ error    = /u/test/mpi_batch.$(Process).err
#@ output   = /u/test/mpi_batch.$(Process).out
#@ wall_clock_limit = 00:05:00, 00:04:15
```

To run all 16 tasks of a LAPI program (**hw_r_lapi**) on the **c132f1rp01** node, using User Space and a single network's windows, you could use the following LoadLeveler job command file. The output and error files are created with names that include the process ID.

```
#!/bin/ksh
#@ job_type=parallel
#@ environment = COPY_ALL; \
MP_PROCS=16
#@ requirements = ( Machine == "c132f1rp01" )
#@ executable = /usr/bin/poe
#@ arguments = hw_r_lapi -labelio yes -procs 4
#@ error    = /u/voe3/lapi_batch.$(Process).err
#@ output   = /u/voe3/lapi_batch.$(Process).out
#@ wall_clock_limit = 00:05:00, 00:04:15
# @ network.lapi = sn_single,shared,us
# @ queue
```

For more information, refer to *Tivoli Workload Scheduler LoadLeveler: Using and Administering*.

Submitting an interactive POE job using a TWS LoadLeveler command file

POE users may specify a LoadLeveler job command file to be used for an interactive job.

Using a LoadLeveler job command file provides the capability to:

- Exploit new or existing LoadLeveler functionality that is **not available** using POE options. This includes specification of:
  - task geometry
  - blocking factor
  - machine order
  - consumable resources
  - memory requirements
  - disk space requirements
  - machine architecture

  For more information on the LoadLeveler functionality you can exploit, refer to For more information, see *Tivoli Workload Scheduler LoadLeveler: Using and Administering*
- Run parallel jobs without specifying a host file or pool, thereby causing LoadLeveler to select nodes for the parallel job from any in its cluster.
- Specify that a job should run from more than 1 pool.

You can use a LoadLeveler job command file with or without a host list file. If you have created a LoadLeveler job command file for node allocation (either independently or in conjunction with a host list file), you need to set the **MP_LLFILE** environment variable (or use the **-llfile** flag when invoking the program) to specify the file. You can specify the LoadLeveler job command file using its relative or full path name.

Table 46 describes how to set the **MP_LLFILE** environment variable and the **-llfile** command line flag.

For example, say the LoadLeveler job command file is named *file.cmd* and is located in the directory */u/dlecker*. You could:

Table 46. Example of setting the MP_LLFILE environment variable or -llfile command line flag

Set the MP_LLFILE environment variable:	Use the -llfile flag when invoking the program:
ENTER **export MP_LLFILE=/u/dlecker/file.cmd**	**ENTER** **poe** *program* **-llfile /u/dlecker/file.cmd**

When the **MP_LLFILE** environment variable, or the **-llfile** command line option is used, the following POE node/adapter specifications are ignored.

- **MP_ADAPTER_USE**
- **MP_CPU_USE**
- **MP_DEVTYPE**
- **MP_EUIDEVICE**
- **MP_EUILIB**
- **MP_INSTANCES**
- **MP_MSG_API**

- **MP_NODES**
- **MP_PROCS** (when a host list file is not used.)
- **MP_RDMA_COUNT** (PE for AIX only)
- **MP_RESD**
- **MP_RMPOOL**
- **MP_TASK_AFFINITY**
- **MP_TASKS_PER_NODE**
- **MP_USE_BULK_XFER**

Note also that if the LoadLeveler job command file contains the **#@environment** keyword, none of the environment variable settings within that string will have an effect on the POE or remote task environments.

When using this option, the following restrictions apply.
- Cannot be used for batch POE jobs.
- The host list file cannot contain pool requests.
- The **MP_PROCS** environment variable or the **-procs** command line flag must be used if a host list file is used, otherwise only 1 parallel task will be run on the first host listed in the host list file.
- Certain LoadLeveler keywords are not allowed in the LoadLeveler job command file when it is being used for an interactive POE job. For more information, see *Tivoli Workload Scheduler LoadLeveler: Using and Administering* for a listing of these keywords.

Generating an output TWS LoadLeveler job command file

When using LoadLeveler for submitting an interactive job, you can, provided you are not already using a LoadLeveler job command file, generate an output LoadLeveler job command file. This output LoadLeveler job command file contains the LoadLeveler settings that result from the environment variables and/or command line options for the current invocation of POE. If you are unfamiliar with LoadLeveler and its job command files, this provides an easy starting point for creating LoadLeveler job command files. Once you create an output LoadLeveler job command file, you can then, for subsequent submissions, modify it to contain additional LoadLeveler specifications (such as new LoadLeveler functionality available only through using a LoadLeveler job command file).

Be aware that you cannot generate a LoadLeveler job command file if you are already using one; in other words, if the **MP_LLFILE** environment variable or the **-llfile** command line flag is used. You also cannot generate an output LoadLeveler job command file if you are submitting a batch job.

To generate a LoadLeveler job command file, you can use the **MP_SAVE_LLFILE** environment variable to specify the name that the output LoadLeveler job command file should be saved as. You can specify the output LoadLeveler job command file name using a relative or full path name. As with most POE environment variables, you can temporarily override the value of **MP_SAVE_LLFILE** using its associated command line flag **-save_llfile**. Table 47 on page 78 describes how to set the **MP_SAVE_LLFILE** environment variable and the **-save_llfile** command line flag.

For example, to save the output LoadLeveler job command file as **file.cmd** in the directory **/u/wlobb**, you could:

Table 47. Example of setting the MP_SAVE_LLFILE environment variable or -save_llfile command line flag

Set the MP_SAVE_LLFILE environment variable:	Use the -save_llfile flag when invoking the program:
ENTER **export MP_SAVE_LLFILE=/u/wlobb/file.cmd**	**ENTER** **poe** *program* **-save_llfile /u/wlobb/file.cmd**

Parallel file copy utilities

During the course of developing and running parallel applications on numerous nodes, the potential need exists to efficiently copy data and files to and from a number of places.

POE provides three utilities for this reason:

1. **mcp** - to copy a single file from the home node to a number of remote nodes. This was discussed briefly in "Step 2: Copy files to individual nodes" on page 15.
2. **mcpscat** - to copy a number of files from task 0 and scatter them in sequence to all tasks, in a round-robin order.
3. **mcpgath** - to copy (or gather) a number of files from all tasks back to task 0.

mcp is for copying the same file to all tasks. The input file must reside on task 0. You can copy it to a new name on the other tasks, or to a directory. It accepts the source file name and a destination file name or directory, in addition to any POE command line argument, as input parameters.

mcpscat is intended for distributing a number of files in sequence to a series of tasks, one at a time. It will use a round-robin ordering to send the files in a one to one correspondence to the tasks. If the number of files exceeds the number of tasks, the remaining files are sent in another round through the tasks.

mcpgath is for when you need to copy a number of files from each of the tasks back to a single location, task 0. The files must exist on each task. You can optionally specify to have the task number appended to the file name when it is copied.

Both **mcpscat** and **mcpgath** accept the source file names and a destination directory, in addition to any POE command line argument, as input parameters. You can specify multiple file names, a directory name (where all files in that directory, not including subdirectories, are copied), or use wildcards to expand into a list of files as the source. Wildcards should be enclosed in double quotes, otherwise they will be expanded locally, which may not produce the intended file name resolution.

These utilities are actually message passing applications provided with POE. Their syntax is described in Chapter 6, "Parallel Environment commands," on page 129.

Considerations for using the High Performance Switch interconnect

The High Performance Switch supports dedicated User Space (US) and IP sessions, running concurrently on a single node. Users of IP communication programs that are not using LoadLeveler may treat these adapters like any other IP-supporting adapter.

IP message passing programs may or may not use LoadLeveler to allocate nodes, but User Space message passing programs must use LoadLeveler to allocate nodes. When using LoadLeveler, nodes may be requested by name or number from one system pool only. When specifying node pools, the following rules apply:

- All the nodes in a pool should support the same combination of IP and User Space protocols. In other words, all the nodes should be able to run:
  - the IP protocol

    or
  - the User Space protocol

    or
  - the IP and User Space protocols concurrently.
- In order to run the IP protocol, the IP switch addresses must be configured and started. For more information regarding these protocols and LoadLeveler, see *Tivoli Workload Scheduler LoadLeveler: Using and Administering* for more information.
- By default, pool requests for the User Space message passing protocol also request exclusive use of the node(s). As long as a node was allocated through a pool request (and not through a specific node request), LoadLeveler will not allocate concurrent IP message passing programs on the node. You can override this default so that the node can be used for both IP and User Space programs by specifying *multiple* CPU usage.
- By default, requests for the IP message passing protocol also request multiple use of the node; LoadLeveler can allocate both IP and User Space message passing programs on this node. You can override this default so that the node is designated for exclusive use by specifying *unique* CPU usage.
- When running a batch parallel program under LoadLeveler, the adapter and CPU are allocated as specified by the *network keyword* in the LoadLeveler Job Command File, which can also include the specifications for multiple adapters and striping. See *Tivoli Workload Scheduler LoadLeveler: Using and Administering* for more information.
- To use the InfiniBand interconnect, it is recommended that you specify a configured adapter name with **MP_EUIDEVICE**, and set the **MP_DEVTYPE** environment variable to *ib*.

  Note: InfiniBand is not supported on the System x platform.

Scenario 1: Explicitly allocating nodes with TWS LoadLeveler

PE for Linux users: Since the InfiniBand interconnect is the only User Space switch type that is supported with PE for Linux, this example assumes you have set the **MP_DEVTYPE** environment variable to *ib*, or specified the **-devtype** flag as **-devtype ib**.

A POE user, Paul, wishes to run a User Space job 1 in nodes A, B, C, and D. He doesn't mind sharing the node with other jobs, as long as they are not also running in US. To do this, he specifies **MP_EUIDEVICE=sn_single MP_EUILIB=us**, **MP_PROCS=4**, **MP_CPU_USE=multiple**, and **MP_ADAPTER_USE=dedicated**. In his host file, he also specifies:

```
node_A
node_B
node_C
node_D
```

The POE Partition Manager (PM) sees that this is a User Space job, and asks LoadLeveler for dedicated use of the adapter on nodes A, B, C, and D and shared use of the CPU on those nodes. LoadLeveler then allocates the nodes to the job, recording that the sn_single/US session on A, B, C, and D has been reserved for dedicated use by this job, but that the node may also be shared by other users.

While job 1 is running, another POE user, Dan, wants to run another User Space job, job 2, on nodes B and C, and is willing to share the nodes with other users. He specifies **MP_EUIDEVICE=sn_single**, **MP_EUILIB=us**, and **MP_PROCS=2**, **MP_CPU_USE=multiple**, and **MP_ADAPTER_USE=dedicated**. In his host file, he also specifies:

```
node_B
node_C
```

The PM, as before, asks LoadLeveler for dedicated use of the adapter on nodes B and C. LoadLeveler determines that this adapter has already been reserved for dedicated use on nodes B and C, and does not allocate the nodes again to job 2. The allocation fails, and POE job 2 cannot run.

While job 1 is running, a second POE user, John, wishes to run IP/switch job 3 on nodes A, B, C, and D, but doesn't mind sharing the node and the High Performance Switch with other users. He specifies **MP_EUIDEVICE=sn_single** , **MP_EUILIB=ip**, **MP_PROCS=4**, **MP_CPU_USE=multiple**, and **MP_ADAPTER_USE=shared**. In his host file, he also specifies;

```
node_A
node_B
node_C
node_D
```

The POE PM asks LoadLeveler, as requested by John, for shared use of the adapter and CPU on nodes A, B, C, and D. LoadLeveler determines that job 1 permitted other jobs to run on those nodes as long as they did not use the sn_single/US session on them. The allocation succeeds, and POE IP/switch job 3 runs concurrently with POE User Space job 1 on A, B, C, and D.

The scenario above, illustrates a situation in which users do not mind sharing nodes with other users' jobs. If a user wants his POE job to have dedicated access to nodes or the adapter, he would indicate that in the environment by setting **MP_CPU_USE=unique** instead of **multiple**. If job 1 had done that, then job 3 would not have been allocated to those nodes and, therefore, would not have been able to run.

Scenario 2: Implicitly allocating nodes with TWS LoadLeveler

PE for Linux users: Since the InfiniBand interconnect is the only User Space switch type that is supported with PE for Linux, this example assumes you have set the **MP_DEVTYPE** environment variable to *ib*, or specified the **-devtype** flag as **-devtype ib**.

In this scenario, all nodes have both sn_single/US and sn_single/ip sessions configured, and are assigned to pool 2.

In this example, we have eight nodes; A, B, C, D, E, F, G, H.

Job 1: Job1 is interactive, and requests 4 nodes for User Space using **MP_RMPOOL**.

```
MP_PROCS=4

MP_RMPOOL=2

MP_EUILIB=us
```

LoadLeveler allocates nodes A, B, C, and D for dedicated adapter (forced for US) and dedicated CPU (default for MP_RMPOOL).

Job 2: Job 2 is interactive, and requests six nodes for User Space using host.list.

```
MP_PROCS=6

MP_HOSTFILE=./host.list

MP_EUILIB=us

MP_CPU_USE=multiple
MP_ADAPTER_USE=shared
host.list

     @2
```

POE forces the adapter request to be dedicated, even though the user specified shared. Multiple (shared CPU) is supported, but in this case LoadLeveler doesn't have six nodes, either for CPU or for adapter, so the job fails.

Job 3: Job 3 is interactive and requests six nodes for IP using **MP_RMPOOL**.

```
MP_PROCS=6

MP_RMPOOL=2

MP_EUILIB=ip
```

The defaults are shared adapter and shared CPU, but LoadLeveler only has four nodes available for CPU use, so the job fails.

Job 4: Job 4 is interactive and requests three nodes for IP using **MP_RMPOOL**.

```
MP_PROCS=3

MP_RMPOOL=2

MP_EUILIB=ip
```

The defaults are shared adapter and shared CPU. LoadLeveler allocates nodes E, F, and G.

Job 5: Job 5 is interactive and requests two nodes for IP using **MP_RMPOOL**.

```
MP_PROCS=2

MP_RMPOOL=2

MP_EUILIB=ip
```

The defaults are shared adapter and shared CPU. LoadLeveler allocates two nodes from the list E, F, G, H (the others are assigned as dedicated to job 1).

Scenario 3: Implicitly allocating nodes with TWS LoadLeveler (mixing dedicated and shared adapters)

PE for Linux users: Since the InfiniBand interconnect is the only User Space switch type that is supported with PE for Linux, this example assumes you have set the **MP_DEVTYPE** environment variable to *ib*, or specified the **-devtype** flag as **-devtype ib**.

In this scenario, all nodes have both sn_single/US and sn_single/ip sessions configured, and are assigned to pool 2.

In this example, we have eight nodes; A, B, C, D, E, F, G, H

Job 1: Job 1 is interactive and requests four nodes for User Space using host.list.

```
MP_PROCS=4

MP_HOSTFILE=./host.list

MP_EUILIB=us

MP_CPU_USE=multiple
MP_ADAPTER_USE=dedicated
host.list

     @2
```

LoadLeveler allocates nodes A, B, C, and D for dedicated adapter (forced for US), and shared CPU.

Job 2: Job 2 is interactive and requests six nodes for User Space using host.list.

```
MP_PROCS=6

MP_HOSTFILE=./host.list

MP_EUILIB=us

MP_CPU_USE=multiple
MP_ADAPTER_USE=shared
host.list

     @2
```

POE forces the adapter request to be dedicated, even though the user has specified shared. Multiple (shared CPU) is supported, but in this case, LoadLeveler doesn't have six nodes for the adapter request, so the job fails.

Job 3: Job 3 is interactive and requests six nodes for IP using **MP_RMPOOL**.

```
MP_PROCS=6

MP_HOSTFILE=NULL

MP_EUILIB=ip

MP_RMPOOL=2
```

The defaults are shared adapter and shared CPU. LoadLeveler allocates six nodes for IP from the pool.

Job 4: Job 4 is interactive and requests three nodes for IP using **MP_RMPOOL**.

```
MP_PROCS=3

MP_HOSTFILE=NULL

MP_EUILIB=ip

MP_RMPOOL=2
```

The defaults are shared adapter and shared CPU. LoadLeveler allocates three nodes from the pool.

Considerations for failover and recovery with PE

LAPI provides facilities for higher availability and recovery from link and adapter failures. LAPI can quickly determine when an adapter no longer has the ability to communicate, and as a result will *fail over* and recover all communication on an alternate path. Note, however, that failover and recovery are only supported with running over the User Space protocol, and when running jobs across multiple networks.

These instructions are LAPI-oriented, but are included here to provide information you may find valuable. If you are interested in more specific details about failover and recovery operations, refer to *RSCT: LAPI Programming Guide*.

Failover and recovery

LAPI's failover and recovery function consists of two elements:

1. Monitoring and receiving notification about the communication status of the IBM High Performance Switch (HPS) adapters (AIX) or InfiniBand adapters (AIX or Linux).
2. The use of multiple adapters for redundancy, to enable failover. This element depends on LoadLeveler, with corresponding POE functions that serve as a wrapper to convey requests to LoadLeveler.

Failover and recovery cannot be provided for a job if either of these elements is absent.

Requesting the use of multiple adapters

You can use POE environment variables or LoadLeveler job control file (JCF) keywords to request the use of multiple adapters.

Using POE environment variables to request the use of multiple adapters:

If your job is to survive an adapter failure, there must be some redundancy. As a result, each task of the job needs to be allocated communication instances across at least two different IBM High Performance Switch (HPS) adapters or InfiniBand adapters. An *instance* is an entity that is required for communication over an adapter device. In the user space (US) communication mode, which is specified by setting **MP_EUILIB** to **us**, an instance corresponds to an adapter window. On the other hand, note that in the IP communication mode, which is specified by setting **MP_EUILIB** to **ip**, an instance corresponds to the IP address of a given adapter to be used for communication.

Depending on the number of networks in the system and the number of adapters each node has on each of the networks, you can request the allocation of multiple instances for your job tasks by using a combination of the POE environment variables **MP_EUIDEVICE** and **MP_INSTANCES**. The distribution of these requested instances among the various an IBM Power Systems HPS adapters or InfiniBand networks on the nodes is done by LoadLeveler. Depending on the resources available on each of the adapters and whether the job is using user space or IP, LoadLeveler tries to allocate these instances on different adapters.

To request the use of multiple instances on a system where all nodes have adapters on each of the *n* networks in the system, you can set **MP_EUIDEVICE** to the value **sn_all**. This setting translates to a request for the default number of instances (**1**) from adapters on each of the networks in the system, and a request for a total of *n* instances for each of the job tasks. You do not have to set the **MP_INSTANCES** environment variable. If **MP_EUIDEVICE** is set to **sn_all** and you do set the **MP_INSTANCES** variable to a value *m* (where *m* is a number from **1** through the value of the case-insensitive string **max**), this translates to a request of *m* instances from each of the networks in the system for each job task. For user space, this corresponds to a request for (*m* * *n*) different windows for each job task. For users running over IP, this corresponds to a request for the same number of IBM Power Systems HPS or InfiniBand IP devices.

You must take the following considerations into account while defining the number of instances to use and the value specified for **MP_EUIDEVICE**:

- If *m* is greater than the number of adapters a node has on one of the networks, multiple windows will be allocated from some of the adapters. For users running over IP, the same adapter device will be allocated multiple times.
- LoadLeveler translates the value **max** as a request to allocate the number of instances (as specified by the *max_protocol_instances* variable) that are defined for this job class in the LoadLeveler **LoadL_admin** file. See *Tivoli Workload Scheduler LoadLeveler: Using and Administering* for more information. If you request more instances than the value of *max_protocol_instances*, LoadLeveler allocates a number of instances that is equal to the value of *max_protocol_instances*. To have your job use all adapters on the system across all the networks, you can have the administrator set *max_protocol_instances* for your job class to the number of adapters each node has on each network (assuming that each node has the same number of adapters on each network), and then run your job with **MP_EUIDEVICE** set to **sn_all** and **MP_INSTANCES** set to **max**.
- On a system where every node is connected to more than one common network, setting **MP_EUIDEVICE** to **sn_all** is sufficient to allocate instances from distinct adapters for all job tasks. You do not need to set **MP_INSTANCES**. This is because an adapter is connected to exactly one network, this is a request for instances from each network, and if the request is satisfied, at least two distinct adapters have been allocated for each of the job tasks. In the case of user space, if all windows on the adapters of one or more networks are all used up, the job will not be scheduled until windows are available on adapters of each network.

To request the use of multiple instances on a system where all nodes are connected to a single IBM Power Systems HPS adapter or InfiniBand adapter, or where nodes are connected to multiple networks, but you want your tasks to use adapters that are connected to only one of those networks, you can set **MP_EUIDEVICE** to **sn_single** and **MP_INSTANCES** to *m*, where *m* is a number from **1** through the value of the (case-insensitive) string **max**. This translates to a request for *m* instances on one network only; not, as in the previous case, on each of the *n* networks in the system. With such a request, if **MP_EUILIB** is set to **us**, LoadLeveler may not allocate the multiple windows from distinct adapters if window resources on some of the adapters are consumed by previously-scheduled jobs. In this scenario, LoadLeveler may allocate the multiple windows from a single adapter and one or more of the job tasks will be without a redundant adapter to which they can fail over in the case of a communication problem. As a result, to guarantee that multiple adapters are allocated to the job, and to satisfy the basic requirements for LAPI's failover and recovery function, you must do the following:

1. Have the nodes in the system connect to multiple IBM Power Systems HPS adapters or InfiniBand adapters.
2. Set **MP_EUIDEVICE** to **sn_all**.

POE posts an attention message stating that failover and recovery operations may not be possible for the job if multiple instances are requested, but one or more job tasks are allocated instances that are all from the same adapter. Table 48 shows the interaction among the values of **MP_INSTANCES**, **MP_EUIDEVICE**, and **MP_EUILIB**, in terms of the total instances that are allocated to every task of the job, and whether use of the failover and recovery function is possible as a result.

Table 48. Failover and recovery operations

	Instances allocated per task with MP_EUILIB=us		**Instances allocated per task with MP_EUILIB=ip**	
MP_EUIDEVICE=	**MP_INSTANCES is not set**	**MP_INSTANCES=*m***	**MP_INSTANCES is not set**	**MP_INSTANCES=*m***
sn_single	**1** no failover	*m* **For AIX users:** failover may not be possible. **For Linux users:** Not supported	**1** **For AIX users:** No failover **For Linux users:** Not supported	*m* **For AIX users:** Failover is possible if *num_adapters* per network > **1** **For Linux users:** Not supported
sn_all	*num_networks* failover is possible if *num_networks* > **1**	*m * num_networks* failover is possible if *num_networks* > **1**	*num_networks* **For AIX users:** Failover is possible if *num_networks* > **1** **For Linux users:** Not supported	*m * num_networks* **For AIX users:** Failover is possible if *num_networks* > **1** **For Linux users:** Not supported

Using TWS LoadLeveler JCF keywords to request the use of multiple adapters:

The use of the LoadLeveler job class attribute *max_protocol_instances* is described in "Using POE environment variables to request the use of multiple adapters" on page 83. Although more than eight instances are allowed using a combination of the *max_protocol_instances* setting and the **MP_INSTANCES** environment variable, LAPI ignores all window allocations beyond the first eight, because LAPI supports a maximum of eight adapters per operating system instance and the best performance can be obtained with one window on each of them. Using multiple windows on a given adapter provides no performance advantage.

For more information about the *max_protocol_instances* attribute, and for the syntax to specify the request for multiple instances on a single network or on all networks in the system using a LoadLeveler job control file (JCF), see *Tivoli Workload Scheduler LoadLeveler: Using and Administering*.

Failover and recovery restrictions

- Requesting the use of multiple instances for tasks of the job is for failover/recovery and load balancing among multiple networks only. No performance gain in terms of individual task bandwidth should be expected due to the use of multiple instances.
- Failover and recovery are only supported on snX or InfiniBand adapters. Failover and recovery are not supported for standalone (non-POE) LAPI.
- With PE for AIX, when a job with a failed adapter is preempted, LoadLeveler may not be able to continue with the job if it cannot reload the switch table on the failed adapter. Any adapter failure that causes switch tables to be unloaded will not be recovered during the job run.

Considerations for data striping, with PE

PE MPI depends on LAPI as a lower level protocol and the support for striping is entirely within the LAPI layer. In most cases, the layering of PE MPI on LAPI is transparent to the MPI user. Striping is the distribution of message data across multiple communication adapters in order to increase bandwidth. By using striping in conjunction with the bulk transfer transport mechanism, applications can experience gains in communication bandwidth performance. Applications that do not use the bulk transfer communication mode typically cannot benefit from striping over multiple adapters.

In this case, although the striping implementation is within LAPI, it has implications that affect PE MPI users. These instructions are LAPI-oriented, but are included here to provide information you may find valuable. If you are interested in more specific details about striping, refer to *Reliable Scalable Cluster Technology: LAPI Programming Guide*.

Data striping

When running parallel jobs on processors with IBM High Performance Switches (striping is not supported for InfiniBand interconnects), it is possible to stripe data through multiple adapter windows. This is supported for both IP and User Space protocols.

If the system has more than one switch network, the resource manager allocates adapter windows from multiple adapters. A switch network is the circuit of adapters that connect to the same IBM High Performance Switch. One window is assigned to an adapter, with one adapter each selected from a different switch network.

If the system has only one switch network, the adapter windows are most likely allocated from different adapters, provided that there are sufficient windows available on each adapter. If there are not enough windows available on one of the adapters, the adapter windows may all be allocated from a single adapter.

LAPI manages communication among multiple adapter windows. Using resources that LoadLeveler allocates, LAPI opens multiple user space windows for communication. Every task of the job opens the same number of user space windows, and a particular window on a task can only communicate with the corresponding window on other tasks. These windows form a set of ″virtual networks″, in which each ″virtual network″ consists of a window from each task that can communicate with the corresponding windows from the other tasks. The distribution of data among the various windows on a task is referred to as *striping*, which has the potential to improve communication bandwidth performance for LAPI clients.

To enable striping in user space mode, use environment variable settings that result in the allocation of multiple instances. For a multi-network system, this can be done by setting **MP_EUIDEVICE** to **sn_all**. On a single-network system with multiple adapters per operating system image, this can be done by setting **MP_EUIDEVICE** to **sn_single** and setting **MP_INSTANCES** to a value that is greater than **1**.

For example, on a node with two adapter links, in a configuration where each link is part of a separate network, the result is a window on each of the two networks, which are independent paths from one node to others. For IP communication and for messages that use the user space FIFO mechanism (in which LAPI creates packets and copies them to the user space FIFOs for transmission), striping provides no performance improvement. Therefore, LAPI does not perform striping for short messages, noncontiguous messages, and all communication in which bulk transfer is disabled through environment variable settings.

For large contiguous messages that use bulk transfer, striping provides a vast improvement in communication performance. Bandwidth scaling is nearly linear with the number of adapters (up to a limit of 8) for sufficiently-large messages. This improvement in communication bandwidth stems from: 1) the low overhead needed to initiate the remote direct memory access (RDMA) operations used to facilitate the bulk transfer, 2) the major proportion of RDMA work being done by the adapters, and 3) high levels of concurrency in the RDMA operations for various parts of the contiguous messages that are being transferred by RDMA by each of the adapters.

To activate striping or failover for an interactive parallel job, you must set the **MP_EUIDEVICE** and **MP_INSTANCES** environment variables as follows:

- For instances from multiple networks:

  MP_EUIDEVICE=sn_all — Guarantees that the adapters assigned will be from different networks.
- For instances from a single network:

  MP_EUIDEVICE=sn_single and **MP_INSTANCES=***n* (where *n* is greater than **1** and less than *max_protocol_instances*) — Improved striping performance using RDMA can only be seen if windows are allocated from multiple adapters on the single network. Such an allocation may not be possible if there is only one adapter on the network or if there are multiple adapters, but there are available resources on only one of the adapters.

To activate striping for a parallel job submitted to the LoadLeveler batch system, the network statement of the LoadLeveler command file must be coded accordingly.

- Use this network statement for a LAPI User Space job that uses IBM High Performance Switches on multiple networks:
  ```
  #@ network.lapi = sn_all,shared,us
  ```
- Use this network statement for an MPI and LAPI User Space job that uses IBM High Performance Switches on multiple networks and shares adapter windows:
  ```
  #@ network.mpi_lapi = sn_all,shared,us
  ```

The value of **MP_INSTANCES** ranges from **1** to the maximum value specified by *max_protocol_instances*, as defined in the LoadLeveler **LoadL_admin** file. The default value of *max_protocol_instances* is **1**. See *Tivoli Workload Scheduler LoadLeveler: Using and Administering* for more information.

Communication and memory considerations

Depending on the mode of communication, when multiple IBM Power Systems High Performance Switch (HPS) adapters are used for data striping or for failover and recovery, additional memory or address space resources are used for data structures that are associated with each communication instance. In 32-bit applications, these additional requirements have implications that you must consider before deciding whether to use striping or failover and recovery and the extent to which you will use these functions.

IP communication (PE for AIX only): When multiple IBM Power Systems High Performance Switch (HPS) instances are used for IP communication, LAPI allocates these data structures from the user heap. Some 32-bit applications may therefore need to be recompiled to use additional data segments for their heap by using the **-bmaxdata** compilation flag and requesting a larger number of segments. The default amount of data that can be allocated for 64-bit programs is practically unlimited, so no changes are needed. Alternatively, you can modify the 32-bit executable using the **ldedit** command or by setting the **LDR_CNTRL** environment variable to **MAXDATA**. Base the increase to **-bmaxdata** on what is needed rather than setting it to the maximum allowed (**0x80000000**). Using more segments than required may make certain shared memory features unusable, which can result in poor performance. Also, applications that require the eight allowed segments for their own user data (thus leaving no space for LAPI to allocate structures) must use a single IP instance only (**MP_EUIDEVICE=sn_single**).

For more information about **ldedit**, see *IBM AIX Commands Reference*. For more information about **LDR_CNTRL**, see *IBM AIX Performance Management Guide*.

US communication (PE for AIX only): When multiple IBM Power Systems High Performance Switch (HPS) instances are used for User Space communication, you need to consider the following segment usage information when deciding whether to use striping or failover and recovery. The communication subsystem uses segment registers for several different purposes. The AIX memory model for 32-bit applications uses five segment registers. In a 32-bit executable, there are only 16 segment registers available. In a 64-bit executable, the number of segment registers is essentially unbounded. Because segment registers are abundant in 64-bit job runs, this discussion is important only for 32-bit job runs.

By default, the amount of memory that is available for application data structures (the heap) in a 32-bit job run is somewhat less than 256MB. You can use the compilation flag **-bmaxdata:0x80000000** to allocate 2GB of heap, but this requires eight segment registers. Smaller **-bmaxdata** values use fewer segment registers, but these values limit the size of application data structures. If you try to use every available feature of the communication subsystem and allow 2GB for heap, there will not be enough registers, and your application will lose some performance or perhaps not be able to start.

The segment usage between the IBM High Performance Switch and the InfiniBand interconnect is very different, as shown below.

The IBM High Performance Switch (HPS) communication subsystem uses segments as follows:
- One User Space instance (window): 2
- Each additional instance: 1

- Switch clock: 1
- Shared memory: 1
- Shared memory cross-memory attach: 1

The InfiniBand interconnect uses segments as follows:
- One for the command page mapping: 1
- One for Reliable Connected Queue Pairs when RDMA is used: 1
- A transient segment used during setup: 1
- Shared memory: 1
- Shared memory cross-memory attach: 1

Using MPI and LAPI together with separate windows consumes segments beyond the minimum. Using striping also consumes extra windows. Access to the switch clock for the **MPI_WTIME_IS_GLOBAL** attribute requires a segment register for the High Performance Switch adapter. Turning on **MP_SHARED_MEMORY** requires one segment register for basic functions and a second segment register to exploit cross-memory attach, to accelerate large messages between tasks on the same node. If your application requires a large heap, you may need to forgo some communication subsystem options. For most applications, you can set **MP_CLOCK_SOURCE=AIX** and free one register. If MPI and LAPI calls are used in the application, make sure **MP_MSG_API** is set to **MPI_LAPI** rather than **MPI,LAPI**. Because shared memory uses one pair of registers per protocol, using **MPI_LAPI** rather than **MPI,LAPI** is especially important when combining shared memory and user space. If you do not need to use the striping and failover functions, make sure that **MP_EUIDEVICE** is set to **sn_single** and that **MP_INSTANCES** is not set (in which case, it defaults to **1**) or is set to **1** explicitly.

For 32-bit executables that are compiled to use small pages, the segment registers that are reserved by AIX and by **-bmaxdata** are claimed first. The initialization of user space comes second. If there are not enough registers left, your job will not start. The initialization of shared memory comes last. If there are no registers left, the job will still run, but without shared memory. If there is only one register left, shared memory will be enabled, but the optimization to speed large messages with cross-memory attach will not be used. If there are no registers left, shared memory will be bypassed and on-node communication will go through the network.

For 32-bit executables that use large pages, dynamic segment allocation (DSA) is turned on automatically, so any **-bmaxdata** segments requested are not reserved first for the user heap, but are instead allocated in the order of usage. Thus, if the program allocates memory corresponding to the total size of the requested **-bmaxdata** segments before **MPI_Init** or **LAPI_Init** is called, the behavior would be similar to the small page behavior that is described in the previous paragraph. However, if **MPI_Init** or **LAPI_Init** is called before the memory allocation, segments that were intended for use for the program heap may be first obtained and reserved for windows and for communication library features such as shared memory. In this case, the program will be left with fewer segments to grow the heap than **-bmaxdata** had requested. The program is likely to start by claiming all the segments required for the initialization of the communication subsystem, but will terminate later in the job run on a **malloc** failure as its data structure allocations grow to fill the space that the specified **-bmaxdata** value was expected to provide.

For information about how to use large pages, see *IBM AIX Performance Management Guide*. For information about DSA, see *IBM AIX General Programming Concepts: Writing and Debugging Programs*.

Specifying the format of core files or suppressing core file generation (PE for AIX only)

Using the **MP_COREFILE_FORMAT** environment variable (or its associated command line flag **-corefile_format**), you can determine the format of core files generated when processes terminate abnormally — you can specify either traditional AIX core files or lightweight core files that conform to the Parallel Tool Consortium's *Standardized Lightweight Corefile Format* (LCF).

Table 49 describes how setting the **MP_COREFILE_FORMAT** environment variable or the **-corefile_format** command line flag determines the format of the core files that are generated.

Table 49. MP_COREFILE_FORMAT settings

If the MP_COREFILE_FORMAT environment variable or -corefile_format flag:	**Then:**	**For more information, see:**
is not set/used	standard AIX core files will be generated when processes terminate abnormally.	"Generating standard AIX core files (PE for AIX only)"
specifies the string "STDERR"	the core file information will be output to standard error when processes terminate abnormally.	"Writing core file information to standard error (PE for AIX only)" on page 91
specifies any other string	lightweight core files will be generated when processes terminate abnormally.	"Generating lightweight core files (PE for AIX only)" on page 91

Note: Although the AIX operating system provides its own lightweight core file subroutine and environment variable (**LIGHTWEIGHT_CORE**), be aware that it is intended for serial programs only. When using the AIX **LIGHTWEIGHT_CORE** environment variable with parallel programs compiled with the POE compiler scripts, the resulting output is unpredictable. For this reason, you should use the POE lightweight core file flags and environment variables for parallel programs.

Generating standard AIX core files (PE for AIX only)

By default, POE processes that terminate abnormally generate standard AIX core files. Since this is the default behavior, you will not typically need to explicitly specify that standard AIX core files should be generated. If, however, the **MP_COREFILE_FORMAT** environment variable has previously been set, you will need to unset it in order to once again get the default behavior. To unset the **MP_COREFILE_FORMAT** environment variable, you would

ENTER
    unset MP_COREFILE_FORMAT

Generating core files for sigterm (PE for AIX only)

POE automatically generates core files for those signals that result in core files, with the exception of **SIGTERM**. This is because the **SIGTERM** signal can also be issued as the result of an explicit request to terminate via an MPI_Abort() call, in which case, it may not be beneficial to have a core file created.

POE provides an option, via the **MP_COREFILE_SIGTERM** environment variable (and the corresponding **-corefile_sigterm** command line flag), to allow the creation of a core file for **SIGTERM**, when **MP_COREFILE_SIGTERM** or **-corefile_sigterm** is set to *yes*. The default is *no*.

Writing core file information to standard error (PE for AIX only)

As described in "Generating standard AIX core files (PE for AIX only)" on page 90, POE processes that terminate abnormally will, by default, generate standard AIX core files. If you prefer, you can instruct POE to write the stack trace or lightweight core file information to standard error instead. To do this, set the **MP_COREFILE_FORMAT** environment variable to the string *STDERR* (in uppercase). As with most POE environment variables, you can temporarily override the value of **MP_COREFILE_FORMAT** using its associated command line flag — **corefile_format**. Table 50 describes how to set the **MP_COREFILE_FORMAT** environment variable and the **-corefile_format** command line flag to write core file information to standard error.

For example, to specify that lightweight core file information should be written to standard error, you could:

Table 50. Example of writing core file information to standard error by setting the MP_COREFILE_FORMAT environment variable or -corefile_format command line flag

Set the MP_COREFILE_FORMAT environment variable:	Use the -corefile_format flag when invoking the program:
ENTER **export MP_COREFILE_FORMAT=***STDERR*	**ENTER** **poe** *program* **-corefile_format** *STDERR*

Generating lightweight core files (PE for AIX only)

By default, POE processes that terminate abnormally generate standard AIX core files. Often, however, traditional AIX core files are insufficient for debugging your program. This is because traditional AIX core files provide information that is too low-level for you to get a general picture of the overall status of your program. In addition, traditional AIX core files tend to be large and so can consume too much, if not all, available disk space. In being written out, theses core files can take up an unacceptable amount of CPU time and network bandwidth. These problems are especially acute in a large-scale parallel-processing environment, when the problems can be multiplied by hundreds or thousands of processes.

To address these problems with traditional core files, the Parallel Tools Consortium (a collaborative body of parallel-programming researchers, developers, and users from governmental, industrial, and academic sectors) has developed a core file format called the *Standardized Lightweight Corefile Format* (LCF). As its name implies, a lightweight core file does not have the often unnecessary low-level detail found in a traditional core file; instead a lightweight core file contains thread stack traces (listings of function calls that led to the error). Because of its smaller size, a lightweight core file can be generated without consuming as much disk space, CPU time, and network bandwidth as a traditional AIX core file. In addition, the LCF format can be a more useful aid in debugging threaded programs.

Using the **MP_COREFILE_FORMAT** environment variable (or its associated command line flag **-corefile_format**), you can specify that POE should generate lightweight core files instead of standard AIX core files. To do this, simply specify the lightweight core file name. Table 51 describes how to set the **MP_COREFILE_FORMAT** environment variable and the **-corefile_format** command line flag to specify that POE should generate lightweight core files.

For example, to specify the lightweight core file name *light_core*, you could:

Table 51. Example of specifying lightweight core files by setting the MP_COREFILE_FORMAT environment variable or -corefile_format command line flag

Set the MP_COREFILE_FORMAT environment variable:	Use the -corefile_format flag when invoking the program:
ENTER **export MP_COREFILE_FORMAT=***light_core*	**ENTER** **poe** *program* **-corefile_format** *light_core*

One lightweight core file (in this example, named *light_core*) for each process will be saved in a separate subdirectory.

By default, these subdirectories will be prefixed by the string *coredir* and suffixed by the task id (as in *coredir.0*, *coredir.1*, and so on). You can specify a prefix other than the default *coredir* by setting the **MP_COREDIR** environment variable or **-coredir** flag as described in "Multi-task core file" on page 65.

Note: By setting **-coredir** or **MP_COREDIR** to *none* you can bypass saving lightweight core files in a new subdirectory, and have them saved in /tmp instead.

In addition to developing the LCF standard, the Parallel Tools Consortium has also created command line and graphical user interface tools (not distributed by IBM) that you can use to analyze lightweight core files. To use these tools, you will first want to merge the separate lightweight core files into a single file — with each separate lightweight core file's information appended, one after another, into the single lightweight core file. To merge the separate lightweight core files into a single file, you could, for example, use the **mcpgath** command (as described in "mcpgath" on page 145) or you could create and use your own script.

Note: The lightweight core file stack traces, and, by extension, the lightweight core file browsers, will be able to show source code line numbers only if your program is compiled with the **-g** option. Otherwise, locations will be shown by relative address within the module. The **-g** flag is a standard compiler flag that produces an object file with symbol table references. For more information on the **-g** option, refer to its use on the **cc** command as described in *IBM AIX Version 5: Commands Reference*

Managing large memory parallel jobs (PE for AIX only)

If you submit a job that requires a large amount of paging space, but did not compile it to use large pages, the result can be node instability or even system failure. To avoid these conditions, you can use the **MP_TLP_REQUIRED** environment variable (or **-tlp_required** command line flag) to appropriately respond to jobs that were not compiled for large pages.

When you set **MP_TLP_REQUIRED** to *warn*, POE detects and issues a warning message for any job that was not compiled for large pages. Setting **MP_TLP_REQUIRED** to *kill* causes POE to detect and kill any job that was not compiled for large pages. For more information, see Chapter 7, “POE Environment variables and command line flags,” on page 215.

Running programs under the C shell (PE for AIX only)

During normal configuration, the Automount Daemon (amd) is used to mount user directories. amd’s maps use the symbolic file system links, rather than the physical file system links. While the Korn shell keeps track of file system changes, so that a directory is always available, this mapping does not take place in the C shell. This is because the C shell only maintains the physical file system links.

As a result, users that run POE from a C shell may find that their current directory (for example **/a/moms/fileserver/sis**), is not known to amd, and POE fails with message 0031-214 (unable to change directory).

By default, POE uses the Korn shell **pwd** command to obtain the name of the current directory. This works for C shell users if the current directory is either:

- The home directory
- Not mounted by amd.

If neither of the above are true (for example, if the user’s current directory is a subdirectory of the home directory), then POE provides another mechanism to determine the correct amd name; the **MP_REMOTEDIR** environment variable.

POE recognizes the **MP_REMOTEDIR** environment variable as the name of a command or Korn shell script that echoes a fully-qualified file name. **MP_REMOTEDIR** is run from the current directory from which POE is started.

If you do not set **MP_REMOTEDIR**, the command defaults to **pwd**, and is run as **ksh -c pwd**. POE sends the output of this command to the remote nodes and uses it as the current directory name.

You can set **MP_REMOTEDIR** to some other value and then export it. For example, if you set **MP_REMOTEDIR="echo /tmp"**, the current directory on the remote nodes becomes /tmp on that node, regardless of what it is on the home node.

The script **mpamddir** is also provided in **/usr/lpp/ppe.poe/bin**, and the setting **MP_REMOTEDIR=mpamddir** will run it. This script determines whether or not the current directory is a mounted file system. If it is, the script searches the amd maps for this directory, and constructs a name for the directory that is known to amd. You can modify this script or create additional ones that apply to your installation.

Note: Programs that depend upon the name of the current directory for correct operation may not function properly with an alternate directory name. In this case, you should carefully evaluate how to provide an appropriate name for the current directory on the home nodes.

If you are executing from a subdirectory of your home directory, and your home directory is a mounted file system, it may be sufficient to replace the C shell name of the mounted file system with the contents of $HOME. One approach would be:

```
export MP_REMOTEDIR=pwd.csh
```

or for C shell users:

```
setenv MP_REMOTEDIR pwd.csh
```

where the file **pwd.csh** is:

```
#!/bin/csh -fe

# save the current working directory name

set oldpwd = `pwd`

# get the name of the home directory

cd $HOME

set hmpwd = `pwd`

# replace the home directory prefix with the contents of $HOME

set sed_home = `echo $HOME | sed 's/\//\\\//g'`

set sed_hmpwd = `echo $hmpwd | sed 's/\//\\\//g'`

set newpwd = `echo $oldpwd | sed "s/$sed_hmpwd/$sed_home/"`

# echo the result to be used by amd

echo $newpwd
```

Using RDMA

Remote Direct Memory Access (RDMA) is a mechanism that allows large contiguous messages to be transferred while reducing the message transfer overhead. PE support for RDMA differs, depending on the operating system you are using. PE for AIX supports RDMA on both the IBM High Performance Switch and the InfiniBand interconnect, while PE for Linux supports RDMA on the InfiniBand interconnect only.

Using RDMA with the IBM High Performance Switch (PE for AIX only)

To use RDMA with the IBM High Performance Switch, **MP_USE_BULK_XFER** must be set to **YES**. The default is **NO**. Bulk data transfer is possible only using RDMA. If necessary, **MP_USE_BULK_XFER** can be overridden with the command line option, **-use_bulk_xfer**.

MP_RDMA_COUNT is used to specify the number of user RDMA context (rCxt) blocks (for adapter resources). This number represents the total number of rCxt blocks required by the application program, by determining the number of remote handles the program will require, divided by 128 and adding 2. **MP_RDMA_COUNT** supports the specification of multiple values when multiple protocols are involved. The format can be one of the following:

- **MP_RDMA_COUNT**=**m** for a single protocol
- **MP_RDMA_COUNT**=**m,n** for multiple protocols. Only for when **MP_MSG_API**=**mpi.lapi** – the values are positional, m is for MPI, n for LAPI.

Note that the **MP_RDMA_COUNT**/**–rdma_count** option signifies the number of rCxt blocks the user has requested for the job, and LoadLeveler determines the actual number of rCxt blocks that will be allocated for the job. POE will use the value of **MP_RDMA_COUNT** to specify the number of rCxt blocks requested on the LoadLeveler MPI and/or LAPI network information when the job is submitted. The number of rCxt blocks will be the same for every window of the same protocol.

Applications that set **MP_USE_BULK_XFER** imply that RDMA will be used with a single rCxt block (or an extra, if **MP_RDMA_COUNT** is set), per window. For striping and failover, the same number of rCxt blocks are assigned to each window.

The **MP_RDMA_COUNT** specification only has meaning for LAPI applications. When **MP_RDMA_COUNT** is specified for MPI applications (either when **MP_MSG_API** is explicitly set or defaults to **mpi**), POE will issue a warning message that the **MP_RDMA_COUNT** specification is unnecessary.

The use of the **MP_RDMA_COUNT** specification requires LoadLeveler 3.3.1 or later.

Using RDMA with the InfiniBand interconnect

PE supports RDMA over the InfiniBand interconnect with either the AIX or Linux operating system. In order to support RDMA over Infiniband, PE requires the use of *Reliable Connected Queue Pairs* (RC QPs) to establish adapter resources, and LoadLeveler manages those resources on behalf of the application. POE interacts with LoadLeveler to determine the resources allocated, and then passes that information to MPI and LAPI. To learn more about InfiniBand and Reliable Connected Queue Pairs, you may find the InfiniBand Trade Association Web site (**http://www.infinibandta.org/specs/**) helpful.

LAPI creates the RC QPs for a given pair of tasks, based on the values you provide using various environment variables. The RC QPs that LAPI creates may be used for RDMA communication between that pair of tasks for all future contiguous messages that are larger than the RDMA threshold (which you determine using the **MP_BULK_MIN_MSG_SIZE** environment variable). You can also specify a maximum number of RC QPs that can be created for a task by setting the **MP_RC_MAX_QP** environment variable (the default is no maximum). Limiting the number of RC QPs per task allows you to reserve memory that can be used for computation. The RC QPs exist until the job is terminated, checkpointed, or preempted.

The environment variables used by LAPI to create the RC QPs are described below.

MP_RC_MAX_QP
: Specifies the maximum number of RC QPs that can be created. The allowable value is any positive integer. The default is 2147483647 (which is unlimited). Note that the purpose of **MP_RC_MAX_QP** is to limit the amount of memory that is consumed by RC QPs. It is suggested that you only set this variable if you suspect that your application is performing poorly due to lack of memory.

MP_RC_USE_LMC
: Determines whether LMC (Lid Mask Control) is enabled. Enabling the use of LMC can improve performance, because a single port can support multiple RC paths. The default value is **no** (only one RC connected path is supported). Setting **MP_RC_USE_LMC** to **yes** causes multiple RC paths to be supported, which may improve performance.

LoadLeveler also needs to be aware that a POE job is using bulk transfer (RDMA). LoadLeveler determines this by the value of the LoadLeveler *bulkxfer* keyword, which can be explicitly set to **yes** (**#@ bulkxfer=yes**) in a LoadLeveler JCF file, or by setting the POE **MP_USE_BULK_XFER** environment variable.

The administrator must perform the following tasks to enable the use of RDMA:

- Set the **SCHEDULE_BY_RESOURCES = RDMA** keyword, in the LoadLeveler configuration file. **SCHEDULE_BY_RESOURCES** specifies the consumable resources that are considered by the LoadLeveler schedulers. For more information, see *IBM LoadLeveler: Using and Administering*.

  Note that you can confirm which nodes have been enabled by using the LoadLeveler command **llstatus -R**. In the following example output for the **llstatus -R** command, the f4rp02 node is not enabled for RDMA:

  ```
  a [f4rp02]kgoin>llstatus -R

  Machine                 Consumable Resource(Available, Total)
  ----------------------  -----------------------------------------------
  f3rp01.ppd.pok.ibm.com  RDMA(4,4)+<
  f3rp02.ppd.pok.ibm.com
  f4rp03.ppd.pok.ibm.com  suiteshare(16,16) RDMA(4,4)+<
  f4rp04.ppd.pok.ibm.com  RDMA(4,4)+<

  Resources with "+" appended to their names have the Total value reported from
  Startd.
  Resources with "<" appended to their names were created automatically.
  a [f4rp02]kgoin>
  ```

After the administrator has enabled RDMA, users must perform the following tasks:

- Verify that **MP_DEVTYPE** is set to **ib**.
- Request the use of bulk transfer by doing one of the following:

  If you are an interactive user, set the **MP_USE_BULK_XFER** environment variable to **yes**:

  ```
  MP_USE_BULK_XFER=yes
  ```

  The default setting for **MP_USE_BULK_XFER** is **no**.

  If you are a batch JCF user, specify:

  ```
  #@ bulkxfer = true
  ```
- Set the minimum message length for bulk transfer with the **MP_BULK_MIN_MSG_SIZE** environment variable. Contiguous messages with data lengths greater than or equal to the value you specify for this environment

variable use the bulk transfer path. Messages that are noncontiguous or have data lengths that are smaller than the value you specify for this environment variable use the Unreliable Datagram (UD) packet mode method of transfer.
- Set the **MP_RC_MAX_QP** and **MP_RC_USE_LMC** environment variables, as appropriate for your installation.

Improving application scalability performance (PE for AIX only)

There are certain highly-tuned, fine-grained MPI parallel applications that may benefit from using special tuning and dispatching capabilities provided by AIX and Parallel Environment, particularly in system and application environments where scalability and performance are important concerns.

Two features that are available for such applications are:
- POE priority adjustment coscheduler
- AIX Dispatcher tuner

Interaction is required on the part of the system administrator to assess the overall need and options available through these features, and to make them available for general users. With high-computing performance environments, there are certain issues to be considered, based on a variety of factors, some of which may require selecting kernel options that require a system reboot or using workload balancing to dedicated processors for offloading critical system activity.

Users may wish to consult with their system administrator about allowing certain options to be made available to them for their needs. Such options and factors should be carefully weighed and evaluated when using these capabilities.

POE priority adjustment coscheduler

Certain applications can benefit from enhanced dispatching priority (coscheduling) during execution. POE provides a service for periodically adjusting the dispatch priority of a user's task between set boundaries, giving the tasks improved execution priority.

The PE coscheduler works by alternately, and synchronously, raising and lowering the dispatch priority of the tasks in an MPI job. The objective is for all the tasks to have the same priority across all processors, and to force other system activity into periodic and aligned time slots during which the MPI tasks do not actively compete for CPU resources.

When the **MP_PRIORITY** environment variable is specified, POE attempts to use the coscheduler to adjust the priority of the tasks, based on the values specified and the constraints defined by the system administrator. The value of the **MP_PRIORITY** environment variable can be specified in one of two forms:
- A job class, which defines the priority adjustment values
- A list of priority adjustment values, which must fall within predefined limits.

The system administrator needs to define the available constraints and values by defining entries in the **/etc/poe.priority** file. Refer to *IBM Parallel Environment: Installation* for specific information on defining entries in the **/etc/poe.priority** file.

When you specify a *job class* as a value for **MP_PRIORITY**, the specified class must exist in the **/etc/poe.priority** file on each node. POE looks in **/etc/poe.priority** and finds the entry that corresponds to that class, and then uses it to determine the priority adjustment values to be used. The class entry defines the following parameters:

- User name. The user name can also be in the form of an asterisk (wildcard).
- Class name. When a wildcard is used, the class can be used to define a minimum or maximum class threshold.
- High priority (more favored).
- Low priority (less favored).
- Percentage of time to run at high priority.
- Duration of adjustment cycle.

When you specify a list of values for **MP_PRIORITY**, you must specify the string as a colon-separated list in the following format:

```
hipriority:lopriority:percentage:period
```

When the value of the **MP_PRIORITY** environment variable is specified as a list of values, it is evaluated against the maximum and minimum settings in the **/etc/poe.priority** file. The values will only take effect under the following conditions:

- When a *maximum* setting is specified in the file, and each value in the environment variable is less than or equal to the corresponding value in the file.
- When a *minimum* setting is specified in the file, and each value in the environment variable is greater than or equal to the corresponding value in the file.

Refer to *IBM Parallel Environment: Installation* for specific and additional details on the format and meaning of these values.

Note: If your cluster does not have a global time source (for example, an HPS switch), software synchronization of the node clocks (for example, NTP) is required. Otherwise, the high-priority and low-priority windows might not be sufficiently aligned, causing the coscheduler to be ineffective.

When using the coscheduler with AIX, you should also consider the following:

- The normal dispatch priority is 60. If both high and low priority are set to values less than 60, a compute-bound job will prevent other users from being dispatched. The dispatch preference goes to the lower number.
- The high priority value must be equal to or greater than 12. If the value is between 12 and 20, the job competes with system processes for cycles, and might disrupt normal system activity.
- If the high priority value is less than 30, keystroke capture will be inhibited during the high priority portion of the dispatch cycle.
- If high priority is less than 16, the job will not be subject to the scheduler during the high priority portion of the cycle.
- The low priority value must be less than or equal to 254.
- If the high priority value is less than (more favored than) the priority of the IBM High Performance Switch fault-service daemon, and if the low priority portion of the adjustment cycle is less than two seconds, then switch fault recovery will be unsuccessful, and the node will be disconnected from the switch.
- The coscheduling facility allows programs using the User Space library to maximize their effectiveness in interchanging data. The process might also be used for programs using IP, either over the switch or over another supported

device. However, if the high priority phase of the user's program is more favored than the network processes (typically priorities 36-39), the required IP message passing traffic might be blocked and cause the program to hang.

- Consult the include file **/usr/include/sys/pri.h** for definitions of the priorities used for normal AIX functions.
- Each node might have a different **/etc/poe.priority** file that defines the scheduling parameters for tasks running on that node.
- The primary performance enhancement is achieved when the user's application can run with minimal interference from the standard AIX daemons running on each node. This is achieved when the user's application is scheduled with a fixed priority that is more favored than the daemon's, which typically run with a priority setting of 60.
- *More favored* priority values are numerically smaller than *less favored* priority values

When using the coscheduler with Linux, you should also consider the following:

- Consult your Linux distribution's scheduling and priority man pages for the recommended values to use with the **sched_setscheduler** function call.
- The default scheduling policy used by the Linux coscheduler is SCHED_RR.
- The POSIX standard values for scheduling priority are 0 to 99. Processes with numerically higher priority values are scheduled before processes with numerically lower priority values. However, this might vary, depending on your Linux distribution. Also, the actual priority values cannot exceed the system-defined values.
- The typical process priority is 0, with a default policy of SCHED_OTHER. This includes user processes and system daemons, which might vary, depending on your Linux distribution and system configuration.

IMPORTANT NOTE:

The coscheduler relies on **pmadjpri**, which is a set-user-on-execution (setuid) binary file, and is owned by the root system user. This allows system administrators to temporarily grant non-root users the ability to modify the priority of the jobs that are executed by POE.

For AIX users, however, it is important to note that the AIX **fpm** security command, when run by a system administrator, disables setuid programs, including **pmadjpri**. As a result, when the **fpm** command has been run, dynamic priority adjustment of jobs is not possible. Note that this limitation only applies when POE is running under LoadLeveler.

In interactive mode (POE is not running under LoadLeveler): Because the POE Partition Manager Daemon (PMD) runs as root, and the **pmadjpri** program is executed by the PMD, **pmadjpri** will have the necessary authority to adjust priorities normally.

In non-interactive mode (POE is running under LoadLeveler): Because, under LoadLeveler, the POE Partition Manager Daemon (PMD) does not run as root, the **pmadjpri** program fails when calling setuid(0). End users only see output if **MP_PRIORITY_LOG** is set to **yes**.

The following is an example log file:

```
------------------------------------------------------------
Starting pmadjpri at Fri Mar 16 15:56:22 2007


MP_PRIORITY_NTP set to no, NTP will be stopped (if running).
pmadjpri: setuid(0) failed! errno = 1
-------------------------------------------------------------
```

For AIX users, the coscheduler is designed to work with a globally synchronized external clock, such as the switch clock registers on the IBM High Performance Switch. When the coscheduler is started on a node, it looks for the existence of the switch clock. If one is found, the coscheduler turns off the Network Time Protocol (NTP) daemon, if it is running, and synchronizes the AIX clock seconds with the switch clock seconds. The intent is to globally synchronize the AIX time slices applied to the parallel job. When the job terminates, the NTP daemon is restarted, if it had been turned off. The use of the NTP daemon might be controlled with the **MP_PRIORITY_NTP** environment variable and **-priority_ntp** command line flag.

Status and error messages generated during the priority adjustment process are written to the file **/tmp/pmadjpri.***jobid***.log** (this might also be controlled by the POE **MP_PRIORITY_LOG** environment variable and **-priority_log** command line flag). If you wish to store the file in a location other than **/tmp**, you can specify a different directory with the **MP_PRIORITY_LOG_DIR** environment variable. Also, if you wish to give the file a name other than **pmadjpri.***jobid***.log**, you can do so with the **MP_PRIORITY_LOG_NAME** environment variable. See Chapter 7, "POE Environment variables and command line flags," on page 215 for more information about these environment variables.

Note that ownership of the log file is transferred from **root** to the user that is executing the parallel application.

Also note that any error or diagnostic information from POE's invocation of the priority adjustment function will be recorded in the partition manager log (controlled by the POE **MP_PMDLOG** environment variable and **-pmdlog** command line flag.)

AIX Dispatcher tuning (PE for AIX only)

The coscheduler can be used in conjunction with the AIX Dispatcher functions to optimize the process dispatch and interrupt management in the kernel, to allow fine-grained parallel applications to achieve better performance.

The AIX **schedo** command offers the following options that may be of interest:

- **big_tick_size**, to unstagger (**real-time kernel only**) and reduce the number of physical timer interrupts per second. Increasing the **big_tick_size** increases the interval between activations of the dispatcher, and can reduce the amount of overhead for dispatching.
- **force_grq**, to assign all processes that are not part of the PE/MPI job to the global run queue. This allows all non-MPI activity to compete equally for available CPU resources. Without setting this option, non-MPI processes may queue up for resources on a busy processor, when another processor is idle.

The use of such tunables are only fully effective if the AIX kernel is running with the Real Time option, requiring a system reboot. This is required to produce the interrupts necessary for the coscheduler to modify the priorities, and no longer stagger the interrupts.

Once the **big_tick_size** option is changed, interrupts can no longer be staggered until the system is rebooted, even if **big_tick_size** is reset. In addition, if the real-time kernel is enabled without any change to **big_tick_size**, the interrupts will remain staggered.

Also, using the **force_grq** option could degrade system performance when a system is not dedicated to running a parallel job.

The system administrator must enable or disabled these options as well as perform the necessary system reboot.

For additional details on enabling the coscheduler and AIX dispatcher, see *IBM Parallel Environment: Installation*.

Starting a User Space POE job, using the InfiniBand interconnect, without LoadLeveler (PE for AIX only)

You can use the Network Resource Table (NRT) sample programs (**nrt_api**) to load network tables and start a POE application without using LoadLeveler, or any other resource management tool. These sample programs provide a simple example of how POE-MPI or POE-LAPI User Space jobs can be started without LoadLeveler. Note that the sample programs are only intended for use with the InfiniBand interconnect on AIX.

The NRT sample programs use the Network Resource Table APIs, which are documented in *Reliable Scalable Cluster Technology: NRT API Programming Guide*.

Warning: Be very careful when running the sample code. The system administrator should carefully monitor the use of these programs, particularly **nrt_api**, which may be used to load and unload network tables. It is suggested that you use these programs on a set of nodes that have been set aside for testing purposes only.

RESTRICTIONS:

These sample programs are **samples** only; they are simple programs that are intended to illustrate the ways in which it is possible to take advantage of RSCT's NRT APIs for alternative resource managers. These sample programs should not be used as they are in a production environment. They serve only as a guide for customers to develop and test their own programs that utilize the NRT APIs.

The samples represent one way of utilizing the NRT APIs to load the network tables and run a parallel job, but may not be the best way to do so in all cases.

These samples should not be viewed as intended programming interfaces. Therefore, users cannot expect continued or ongoing support for the sample programs. These samples may be changed or discontinued at any time in the future.

nrt_api includes the following files:

/usr/lpp/ppe.poe/samples/nrt/README.nrt
An information readme file

/usr/lpp/ppe.poe/samples/nrt/nrt_api.c
A program for loading and unloading the network tables

/usr/lpp/ppe.poe/samples/nrt/nrt_run
A script for starting a POE application

/usr/lpp/ppe.poe/samples/nrt/makefile
A makefile for building the **nrt_api** program

Step 1: Compile and install the NRT API sample programs (PE for AIX only)

The system administrator must compile the NRT API sample programs and make them available for general use. The sample code, which is located in the **/usr/lpp/ppe.poe/samples/nrt** directory, includes a C program, a makefile, a shell script, and a readme file. For more information on compiling and installing the sample programs, see *IBM Parallel Environment: Installation*.

Step 2: Construct input data files (PE for AIX only)

The sample programs depend on an input data file that contains node and window information for loading and unloading the network tables. This file should reflect the information that is to be used when running the POE application. It represents a global view of the network table and window data that is required on each node and, in some cases, more than one copy of the input data file may be required (such as when loading windows for multiple adapters or ports). The input data file must be placed in the directory from which you run the **nrt_api** command (in "Step 3: Load the network tables on each node (PE for AIX only)" on page 106).

Note that the format of the input data file is different, depending on whether it is to be used for loading or unloading the tables. As a result, the same input data file cannot be used for both loading and unloading the tables. See "Step 2a: Construct an input data file for loading (PE for AIX only)" on page 103 and "Step 2b: Construct an input data file for unloading (PE for AIX only)" on page 105.

Step 2a: Construct an input data file for loading (PE for AIX only)

You specify the input data file when you run the **nrt_api** command (in "Step 3: Load the network tables on each node (PE for AIX only)" on page 106). The contents of the data input file provide a simple description of the network you wish to load, including the:

- Number of tasks in the job
- Adapter device name
- Window number
- LID
- Port number
- Network ID

The **nrt_api** program expects the input data file to be named **nrt_data** (but you if you wish to use another name, you can modify the sample **nrt_api.c** code).

The format of the **nrt_data** file for loading network tables is shown below. Note that each field must be separated by one or more spaces.

The format of line 1 is:

number_of_tasks *job_key* *adapter_string* *network_id*

where:

- *number_of_tasks* is the number of tasks in the job.
- *job_key* is the integer job key value. This value is supplied to the **NRT_JOB_KEY** environment variable in "Step 4: Run the parallel job under POE (PE for AIX only)" on page 107.
- *adapter_string* is the InfiniBand adapter device name character string. This value is supplied to the **NRT_WINDOW_DATA** environment variable in "Step 4: Run the parallel job under POE (PE for AIX only)" on page 107.
- *network_id* is the 64-bit network ID.

The format of line 2, through the end of the file, is as follows. Note that all of the following values must be entered for each task on a separate line, and the order of tasks must be sequential, starting with task 0.

node_number *window_number* *LID_number* *port_number*

where:

- *node_number* is the integer node number assigned by Reliable Scalable Cluster Technology (RSCT) to the node on which the task will run.
- *window_number* is the integer window number to be assigned for this task.
- *LID_number* is the integer LID.
- *port_number* is the integer InfiniBand port number for the window that is used.

Note: The node number, LID, window number, and port number can be determined by using **nrt_api -s** to query the adapter resources. Also in loading the network table data, the samples define a constant LMC value of zero. If a different value is desired, the samples must be modified to account for different LMC values. This may require you to modify the samples to allow for an LMC value as an additional input parameter when loading the network table data.

An example of a data input file is as follows. For this example:

- The job contains three tasks and runs on three different nodes.
- The job key is 5.
- The network ID is 18338657682652659712.
- Task 0 runs on machine c10n01 with a node number of 1, adapter **iba1**, on window 100. The adapter **iba1** on c10n01 has LID 7 and has a single port (port ID 1).
- Task 1 runs on machine c10n02 with a node number of 2, adapter **iba0**, on window 101. The adapter **iba0** on c10n02 has LID 11 and has a single port (port ID 1).
- Task 2 runs on machine c10n03 with a node number of 3, adapter **iba1**, on window 102. The adapter **iba1** on c10n03 has LID 14 and has a single port (port ID 1).

On the c10n01 machine, the contents of the **nrt_data** file would be as follows:

```
3   5     iba1    18338657682652659712
0  100         7  1
1  101        11  1
2  102        14  1
```

On the c10n02 machine, the contents of the **nrt_data** file would be as follows:

```
3     5     iba0    18338657682652659712
0  100         7  1
1  101        11  1
2  102        14  1
```

On the c10n03 machine, the contents of the **nrt_data** file would be as follows:

```
3     5     iba1    18338657682652659712
0  100         7    1
1  101         11  1
2  102         14  1
```

In the example above, note that the files on each node are almost identical except for the first line. The InfiniBand adapter name is different on the second node. Note also that the tasks must be on the same network in order for the job to successfully run.

Example of loading multiple windows on multiple nodes

The following example illustrates the situation in which multiple windows must be loaded on multiple nodes. In this case, more than one input data file is required and more than one network table must be loaded, on each node.

In this example, the following is expected:

- The job contains 4 tasks, and each node runs 2 tasks. The system uses a single adapter.
- The job key is 6.

- The network ID is 18338657682652659712.
- Tasks 0 and 1 run on node z18n01. The adapter is iba0, and has a LID of 7. Each task requires 2 windows:
  - Task 0 uses windows 110 and 114.
  - Task 1 uses windows 111 and 115.
- Tasks 2 and 3 run on node z18n02. The adapter is iba0, and has a LID of 11. Each task requires 2 windows:
  - Task 2 uses windows 112 and 117.
  - Task 3 uses windows 113 and 116.

The first data file contains the following:

```
4          6      iba0     18338657682652659712
1    110   7        1
1    111   7        1
2    112   11     1
2    113   11     1
```

The second data file contains:

```
4          6      iba0     18338657682652659712
1    114   7        1
1    115   7        1
2    116   11     1
2    117   11     1
```

On each node, you must perform two **nrt_api –l** operations to load the two network tables. This loads the multiple windows on each node, so that each task runs with more than one window. The information in "Step 4: Run the parallel job under POE (PE for AIX only)" on page 107 shows how the job needs to be run after loading multiple windows.

Step 2b: Construct an input data file for unloading (PE for AIX only)

When you unload the network and window information you are required to use a node name instead of a node number. After setting up the input data, you can follow the instructions in "Step 5: Unload the network tables (PE for AIX only)" on page 108.

The **nrt_api** program expects the input data file to be named **nrt_data** (but you if you wish to use another name, you can modify the sample **nrt_api.c** code).

The format of the **nrt_data** file for unloading network tables is shown below. Note that each field must be separated by one or more spaces.

The format of line 1 is:

number_of_tasks   *job_key*   *adapter_string*   *network_id*

where:

- *number_of_tasks* is the integer job key value. This value is supplied to the **NRT_JOB_KEY** environment variable in "Step 4: Run the parallel job under POE (PE for AIX only)" on page 107.

- *job_key* is the integer job key value. This value is supplied to the **NRT_JOB_KEY** environment variable in "Step 4: Run the parallel job under POE (PE for AIX only)" on page 107.
- *adapter_string* is the InfiniBand adapter device name character string. This value is supplied to the **NRT_WINDOW_DATA** environment variable in "Step 4: Run the parallel job under POE (PE for AIX only)" on page 107.
- *network_id* is the 64-bit network ID.

The format of line 2, through the end of the file, is as follows. Note that all of the following values must be entered for each task on a separate line, and the order of tasks must be sequential, starting with task 0.

node_name   window_number   LID_number

where:

- *node_name* is a character string node name, for the node on which the windows were previously loaded.
- *window_number* is the integer window number to be assigned for this task.
- *LID_number* is the integer LID.

Note that while the port number is required for loading the tables, it should not be specified when unloading the tables. Also because you must specify a **node number** when loading the tables and a **node name** when unloading the tables, the same **nrt_data** input file cannot be used for both operations.

Note: The node number, LID, window number, and port number can be determined by using **nrt_api -s** to query the adapter resources.

Step 3: Load the network tables on each node (PE for AIX only)

To load the network tables one each node, run the **nrt_api** command with the **-l** option. The syntax of the **nrt_api** command is as follows:

nrt_api [l] [u] [-s device] [-h]

where:

-l Specifies to load the network resource table on this node.

-u Specifies to unload the network resource table on this node.

-s Displays the network resource table data for the specified adapter device on this node. For more information, see "Displaying adapter device status information (PE for AIX only)" on page 108.

-h Displays usage information.

The **nrt_api** program uses the data provided in the **nrt_data** file along with the **nrt_load_table_rdma** function to load the network table on the node on which it will be run. Descriptions of the return codes from **nrt_load_table_rdma** are displayed. Refer to the header file (**/usr/include/nrt.h**) for more information on the return codes. The **nrt_load_table_rdma** and **nrt_load_window** functions are available from the Network Table API library (**libnrt.a**).

Although the example in "Step 2a: Construct an input data file for loading (PE for AIX only)" on page 103 shows how to set up the **nrt_data** file when using unique nodes for each task, you may wish to run multiple tasks on the same node. Multiple tasks that run on the same node are called **common tasks**. For common tasks, there are additional considerations, in particular, the two following cases:

Case A: All common tasks share the same adapter
: In this case, you only need to load the network table once (one call to **nrt_api -l**). It is important to ensure that each task is assigned a unique window on the adapter, and that all assigned windows are available.

Case B: Multiple adapters will be used
: In this case, you must issue one call to **nrt_api -l** for each adapter you use. It is important to ensure that each task is assigned a unique window on the adapter, and that all assigned windows are available.

Perhaps a simpler way to explain the cases above is to say that one **nrt_data** file and one **nrt_api** load are required for each adapter that is used in the system.

Step 4: Run the parallel job under POE (PE for AIX only)

To run the parallel job under POE, use the **nrt_run** script, as follows:

poe nrt_run *myprog poe_and_prog_options*

where *myprog* is the name of the parallel program. The use of a host list file is required when running a parallel job, and its entries must correspond to the nodes on which the network data has been loaded.

The nrt_run script sets the **MP_MPI_NETWORK** and **MP_LAPI_NETWORK** environment variables for each task and then invokes the parallel program (*myprog)*. The following environment variables must be set before you invoke POE:

MP_RESD
: Set to **no**.

MP_DEVTYPE
: Set to **ib**. Required for selecting InfiniBand interconnect.

MP_MSG_API
: Set to **mpi** or **lapi**. **mpi_lapi** and **mpi,lapi** are not supported by these samples, as written.

NRT_JOB_KEY
: Set to the key used in the **nrt_data** file to load the tables.

NRT_WINDOW_COUNT
: Set to the number of instances used by each task. In the example shown in "Step 3: Load the network tables on each node (PE for AIX only)" on page 106, the value for **NRT_WINDOW_COUNT** would be **1**.

NRT_WINDOW_DATA
: The value you provide should contain the window/adapter data for each task, in the following format:

NRT_WINDOW_DATA=*window,adapter_name:window,adapter_name:window,adapter_name*

In the three-task example shown in "Step 2a: Construct an input data file for loading (PE for AIX only)" on page 103, **NRT_WINDOW_DATA** would be set as follows:

```
NRT_WINDOW_DATA=100,iba1:101,iba0:102,iba1
```

To further illustrate the example of multiple windows in "Step 3: Load the network tables on each node (PE for AIX only)" on page 106, **NRT_WINDOW_COUNT** would be set to **2** and **NRT_WINDOW_DATA** would be set to:

```
NRT_WINDOW_DATA=110,iba0:114,iba0:111,iba0:115,iba0:112,iba0:116,iba1:113,iba0:117,iba1
```

Note that this would require two invocations of **nrt_api -l** on each node after the window information had been obtained on each of those nodes.

Step 5: Unload the network tables (PE for AIX only)

To unload the network tables (as root), run the **nrt_api -u** command on each node, with the **nrt_data** file present. This unloads only the windows that were loaded by the previous call to **nrt_api -l**.

An **nrt_data** file that is used as input has a slightly different format, in that a node name is required while the port number used in loading the table is not required. Because of this, it is suggested that you use different **nrt_data** input files for loading and unloading the tables. See "Step 2b: Construct an input data file for unloading (PE for AIX only)" on page 105 for more information on creating an **nrt_data** file for unloading.

Note: After you load the network tables and set up the environment variables, you can run any number of jobs. You only need to unload the tables after you have completed running all of the jobs.

The samples use the **nrt_unload_window**() function to unload the windows. In some cases, it might be safer to use the **nrt_clean_window**() function in order to guarantee that all running tasks are terminated while the windows are released. Refer to *Reliable Scalable Cluster Technology: NRT API Programming Guide* for more information on the differences and capabilities of the **nrt_unload_window**() and **nrt_clean_window**() functions.

Displaying adapter device status information (PE for AIX only)

The **nrt_api.c** program allows you to display the window information and the status of adapter devices that are on the node from which the **nrt_api** command is run. You do this with the **nrt_api -s** command. The **-s** flag requires a device name string. For example:

```
nrt_api -s iba0
```

The output of **nrt_api -s** is displayed to STDOUT. It is recommended that you redirect the output from the STDOUT pipe into a file or other command for later use. The resulting output displays the node, window, port ID, LID, network ID, window IDs, and states for all available windows for the specified device.

The following is an example of the output produced by **nrt_api -s**:

```
NRT version is 420
Getting status for device iba0
node number: 1
number of ports: 1
port id 0 is 1, lid: 25, lmc: 0, network id: 18338657682652659712
Number of windows reporting status: 128
window ID: 0, user uid: 0, loader pid: 0, bulk transfer 0 -- state: NRT_WIN_AVAILABLE
window ID: 1, user uid: 0, loader pid: 0, bulk transfer 0 -- state: NRT_WIN_AVAILABLE
window ID: 2, user uid: 0, loader pid: 0, bulk transfer 0 -- state: NRT_WIN_AVAILABLE
window ID: 3, user uid: 0, loader pid: 0, bulk transfer 0 -- state: NRT_WIN_AVAILABLE
window ID: 4, user uid: 0, loader pid: 0, bulk transfer 0 -- state: NRT_WIN_AVAILABLE
...
...

window ID: 125, user uid: 0, loader pid: 0, bulk transfer 0 -- state: NRT_WIN_AVAILABLE
window ID: 126, user uid: 0, loader pid: 0, bulk transfer 0 -- state: NRT_WIN_AVAILABLE
window ID: 127, user uid: 0, loader pid: 0, bulk transfer 0 -- state: NRT_WIN_AVAILABLE
```

The **nrt_api -s** command allows you to determine the available windows, which is helpful when you are constructing the **nrt_data** input file to be used for loading and unloading the network table resources.

Chapter 4. Debugging parallel programs using DISH and the PDB debugger

PDB is PE's command line debugger for parallel programs. It works together with the Distributed Interactive Shell (DISH), a tool for launching and managing distributed processes interactively, as well as GDB, the GNU project debugger (for Linux) and dbx, a UNIX-based debugger (for AIX).

All of the commands you use with PDB are either DISH commands, for process management, or GDB or dbx commands, for serial debugging. PDB simply configures DISH to manage distributed GDB or dbx instances, and then hands control over to DISH.

Using the Distributed Interactive Shell

Distributed Interactive Shell (DISH) is an interactive tool that serves as a control center to multiple distributed copies of a client. DISH can be configured into a distributed shell, a parallel debugger, or some other interactive program, depending on the client you choose to use.

For example, Korn Shell (**ksh**) can be launched on different nodes and then controlled by DISH. As a result, a command such as **ls** can be broadcast to all shells and then executed in parallel to produce the file listing. DISH then gathers the shell output and presents you with the results.

A DISH session is composed of one or more DISH consoles and one or more DISH agents. There are three types of DISH agents, which are organized into a tree structure. They are:

Root agent
Relays commands from all connected DISH consoles to the remote agents and reports client output from the remote agents back to the consoles. There is only one root agent per DISH session.

Middle agent
Reports client output to DISH through the root agent. The middle agent is a child of the root agent that resides on the same remote host as the client.

Leaf agent
Reports client output to DISH through the middle and root agents. The leaf agent is a child of the root or middle agent, and resides on the same remote host as the client. The leaf agent is responsible for launching the client instances.

By default, DISH launches the root agent from the host on which DISH is running. The middle agents are launched either by parent middle agents or the root agent. The leaf agents are launched by the root and middle agents, or by a job launcher such as POE.

Note that after initial startup, there is only one DISH console in a session. However, you can add up to 64 consoles after the session is started.

The DISH agents connect to DISH through INET sockets, and are able to spawn multiple client instances or additional DISH agents through UNIX pipes. The location of the DISH console is not important, as long as it is accessible to the DISH agents.

At runtime, the root agent reads two required files, which you create; an instance file and a configuration file. The instance file tells DISH the hosts on which the clients (and, therefore, the leaf agents) will run. The configuration file controls the behavior of DISH by defining session-wide variables, and also provides information about the clients to DISH. The configuration file configures the leaf agents for different types of clients, which allows DISH to act as either a distributed shell or a debugger.

The **dish** and **disha** commands are used for setting up DISH and for controlling its behavior. The **dish** command also includes various subcommands for managing activities on the clients.

Figure 1 provides an overview of the DISH environment.

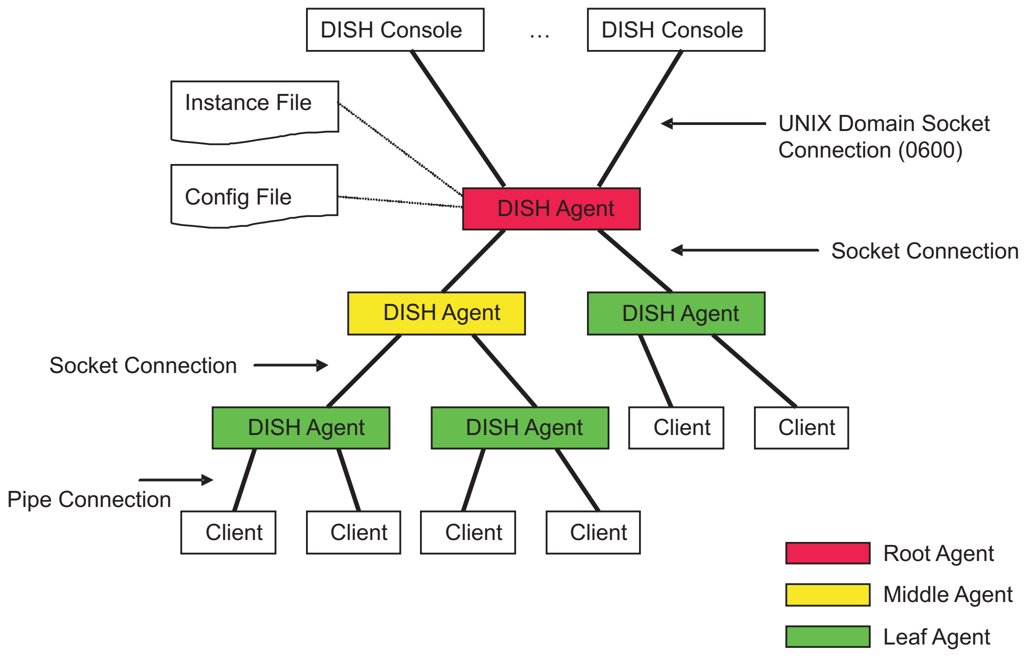

Figure 1. Overview of the DISH environment

Before you begin using DISH

Before using DISH, there are several tasks you need to perform including creating an instance file, creating a configuration file, and testing the clients to make sure they can be supported by DISH.

Creating an instance file

When you set up a DISH session, you need to create an *instance* file to, at a minimum, specify the hosts on which the client instances will run. The instance file defines instance-specific information only, unlike the configuration file, which defines session-wide variables.

Note: If you are using the PDB debugger, you do not need to create the instance file. It is created by PDB dynamically using the POE host list file.

The format of the instance file is similar to a plaintext database, where fields are defined, followed by records consisting of the values for those fields. The first line of the instance file should contain fields for which you will be providing values (*host* and *pid*, in the example below). The values should be entered below the corresponding fields, line-by-line.

As an example, if you wanted to launch DISH with a debugger, you would include the process IDs that are required by DISH in order to attach to a running job. Here is what that instance file might look like:

```
host pid
########################
wcsvr03.rd.acme.com 20396
wcsvr04.rd.acme.com 20379
wcsvr03.rd.acme.com 20397
wcsvr04.rd.acme.com 20380
```

Note that field names in the instance file are case-sensitive.

At a minimum, you must specify a **host** in the instance file for every DISH session. Also note that you can place the **host** variable anywhere in the list of fields. Clients may require other information to be defined in the instance file, but the variable names you add must not conflict with variable names that are already understood by DISH.

host is a reserved variable, which means that it is recognized by DISH, but must be defined by you. It is also the only field that DISH understands. You may add other non-DISH variables, but they are only understood when you specify them using variable substitution. For more information see "Using variable substitution" on page 114.

Note that **id**, **child_pid** and **client_pid** are predefined, instance-specific variables that are also understood by DISH, so you cannot add fields of these names to the instance file. These variables are set implicitly by DISH at run time.

The location of the instance file is specified with the **-I** flag of the **disha** command. For more information about the **disha** command, see "disha" on page 141.

Creating a configuration file

In addition to the instance file, DISH also requires a runtime configuration file to provide information about the client and control the behavior of DISH. It allows DISH to support different client programs without knowing what functions they are performing. Unlike the instance file, which defines information about specific client instances, the configuration file defines session-wide variables.

Note: If you are using the PDB debugger, you do not need to create the configuration file. It is created by PDB.

In the configuration file, you define a list of variables followed by their values (**variable** = *value*). For example:

```
client = /usr/bin/gdb -quiet %exe %pid
client_prompt = (gdb)
interrupt_command = kill -TRAP %pid
```

Both the variables and their values are case-sensitive.

You may add non-DISH variables to the configuration file that are required by the client, but you must specify them using variable substitution. For more information, see “Using variable substitution.”

DISH recognizes the following variables:

- **client** - defines the command line to launch the client.
- **client_prompt** - defines the prompt of the client. The prompt must not change during a DISH session.
- **interrupt_command** - defines the command sequence to be used to interrupt a running client process. When a running client process is successfully interrupted, the prompt should appear again. This command sequence is issued against each client in the current group.
- **key_enter** - for convenience, **key_enter** defines the command to execute when the <Enter> key has been pressed on an empty line.

The location of the configuration file is specified with the **-c** flag of the **disha** command. For more information about the **disha** command, see “disha” on page 141.

Using variable substitution

Clients may require other information to be defined in the instance and configuration files in order to function properly. For example, when a debugger is started, it may require the location of a target program, or it may need to know a process ID to which it should attach. But since DISH knows nothing about the functions of the clients, and only recognizes its own specific fields, you must define all non-DISH variables in the instance file (for instance-specific values) or configuration file (for session-wide values). Note that you may also define non-DISH, session-wide variables by issuing the **disha -g** command from the DISH console.

After the non-DISH values have been defined, you can specify them later using variable substitution. When you specify a value with variable substitution, the value to be substituted is prefixed by a % (percent sign). The value that is used as a substitution is defined in either the instance file or the configuration file.

The following example shows the **client** variable in the configuration file. In this example, GDB (GNU debugger) is the client. The **-quiet** flag suppresses the GDB prolog. The rest of the line asks DISH to substitute *%exe* with the value of the **exe** variable and *%pid* with the value of the **pid** variable.

```
client = /usr/bin/gdb -quiet %exe %pid
```

The **exe** variable is session-wide for SIMD parallel programs, but can also be instance-specific for MIMD parallel programs. The **pid** variable is instance-specific.

Note that you can also use variable substitution with commands you issue from the DISH console. For example, you could issue the following command:

```
attach %pid
```

from the DISH console to cause the GDB or dbx debugger to attach to running processes. *%pid* would be substituted by the value defined for **pid** in the instance file before the command is passed to the GDB or dbx instances. For more information about DISH subcommands, such as attach, see "Subcommands of the dish command" on page 137.

Ensuring the clients are accessible to DISH

After creating the instance and configuration files, you should test the clients to make sure they can be supported by DISH. A quick way to do this is to launch them remotely using **rsh** or **ssh**. If the client's prompt appears, it should work with DISH.

Using the DISH console

As stated earlier, DISH serves as a control center to multiple copies of a client. DISH agents relay commands from the DISH consoles to the client processes, and then report output from the clients back to DISH. The DISH console provides the command line from which you can issue the **dish** and **disha** commands, and also **dish** subcommands, to perform tasks such as:

- Specify the remote shell you want to use for launching clients (**disha -r**)
- Specify whether leaf agents should be launched automatically when DISH is started (**disha -n**)
- Define session-wide variables (**dish -d**)
- Specify an instance file or configuration file other than the default (**disha -t** and **disha -c**)
- Specify a port number to override the default (**disha -p**)

For more information, see "dish" on page 132 and "disha" on page 141.

The **dish** command also includes the following subcommands, which offer additional control over a session:

- **group** - for defining groups of clients
- **on** - for designating a particular group of clients as the current group
- **interrupt** - for interrupting execution of the clients that belong to the current group
- **leave** - for causing the current group to leave the interrupt state
- **kquit** - for killing all running clients and quitting **dish**
- **help** - for getting help information on any of the **dish** subcommands, or on predefined **dish** topics.
- **exit** - for quitting the DISH console.
- **send** - for sending secondary prompt messages to the current group
- **toggle** - for setting PDB's runtime attributes, such as multiple-console broadcast, console message color, and internal message flush.
- **match** - for applying regular expression filter rules to last command output

For more information on the **dish** subcommands see "Subcommands of the dish command" on page 137.

For information about client commands, refer to the client's documentation. To indicate that a command is a client command, you can prefix it with a colon (:). For example, issuing **:help** bypasses DISH and outputs the client's help instead. Client commands are generally broadcast to all members in the current group. A help command with a client command or topic that is not preceded by a colon (:) is sent to only one client.

Setting up the DISH agents

In order to establish a connection to the DISH console, each root, middle, and leaf agent must know the host on which the DISH console is running, and must have an agent ID. To specify the location of the DISH console and the agent ID, you use the **disha** command. The agent ID must be either an integer or the name of an environment variable that specifies the agent ID at run time. Each agent must have a unique agent ID.

Environment variables can also be used to the specify agent IDs. For example, if you are using POE to launch the DISH agents, you can use the **MP_CHILD** environment variable to specify the agent IDs.

For more information about the **disha** command, see "disha" on page 141.

Launching DISH agents

By default, DISH launches the root agent from the host on which DISH is running. The middle agents are launched either by parent middle agents or the root agent. The leaf agents are launched by parent agents, or by a job launcher such as POE.

By default, **rsh** (Remote Shell) is used for launching agents, but if you want to use a different method, you can use the **-r** flag with the **disha** command. For example, if you wanted to use **ssh** (Secure Shell) instead of **rsh**, you would specify **disha -r** *ssh*.

Note that if you have a large number of client sessions, you could experience serious bottlenecks when they are launched. To avoid situations such as this, DISH uses a tree algorithm to cause agents to launch other agents. The root agent generates a tree structure from the instance file and the fanout factor. The fanout factor can be specified with the **-f** option of the **disha** command. The root agent uses **rsh** or **ssh** to launch the next level child agents, and the child agents connect back to the root agent after startup. In turn, the child agents then launch their own child agents.

In addition to DISH, you can also use other job launching mechanisms, for example, POE, to launch DISH agents. Before you do this, you must first tell DISH not to launch leaf agents. You do this with the **disha -n** flag. Next, you need to tell the DISH agents where DISH is running and also supply them with an agent ID for connection. In general, a task ID is available from the job launcher, which can be used as the agent ID.

Understanding the DISH states

At any time, DISH can be in one of three states, in terms of its readiness for input. It is important to be aware of the state, because it indicates what the clients are doing and what commands can be run at that moment. The DISH states are as follows:

- Ready state
  - All clients in the current group have prompted for input.
  - DISH also prompts for input (in other words, by showing *(all)*).
  - Client commands are available. DISH commands, except those for interrupt state only, are also available.
- Execution state
  - At least one client in the current group has not prompted for input.

- DISH does not prompt for input, but shows client output, if any.
- No commands are available, but you can press <Ctrl-C> or send a SIGINT to DISH to get into interrupt state.

- Interrupt state
  - At least one client in the current group has not prompted for input but you pressed <Ctrl-C> or sent a SIGINT.
  - DISH prompts for input and indicates interrupt state (in other words, by showing *(all:int)*). Client output is not displayed.
  - All DISH commands are available but not client commands. Refer to the **interrupt** and **leave** commands, which are available only in interrupt state.

Example: Setting up ksh with DISH

In this example, the user created an instance file with the following contents:

```
client = ksh -i
client_prompt = $
interrupt_command = kill -INT %child_pid
info = echo hostname:pwd
```

The user also created a configuration file with the following contents:

```
host
c121xs03
c121xs04
```

After setting up the instance and configuration files, the user issued the **dish** command, which produced the following output:

```
(all) cd /tmp
(all) %info
0 | c121xs03.ppd.pok.ibm.com:/tmp
1 | c121xs04.ppd.pok.ibm.com:/tmp
(all) ls > ~/tmplist.%id
(all) on 0
(0) ls -l ~/tmplist*
0 | -rw-r--r--  1 user user 293 Aug 25 15:19 /u/user/tmplist.0
0 | -rw-r--r--  1 user user 145 Aug 25 15:19 /u/user/tmplist.1
(0) on all
(all) vmstat 1 3
0 | procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
0 |  r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
0 |  2  0      0 530356  69564 304056    0    0     0     5   17    20  2  4 94  0
1 | procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
1 |  r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
1 |  1  0      0 498456  71868 331132    0    0     0     5   16    18  2  4 94  0
0 |  0  0      0 530356  69564 304056    0    0     0     0 1011    89  0  0 100  0
1 |  0  0      0 498456  71868 331132    0    0     0     0 1009    27  0  0 100  0
0 |  0  0      0 530356  69564 304056    0    0     0     0 1007    24  0  0 100  0
1 |  0  0      0 498456  71868 331132    0    0     0     0 1005    34  0  0 100  0
(all) exit
CONSOLE: 2800-801  Connection terminated.
```

Using the PDB debugger

Debugging with PDB is similar to using multiple GDB or dbx instances to debug multiple processes simultaneously. PDB adds convenience by providing a control center for managing processes (DISH). PDB allows you to debug programs in two different modes; *launch mode* and *attach mode*.

Note that PDB creates the instance file and configuration file dynamically, using the POE host list file. So, although you must create the instance and configuration files when plugging in a shell as a client, you do not need to do so when the client is the PDB debugger.

Before you can use PDB, you first need to compile the program you want to debug. You also need to be able to use POE to launch a parallel job with the compiled program.

Using PDB in launch mode

In launch mode, PDB launches the job and starts interactive debugging from the beginning of the program. To use PDB in launch mode, precede the executable in your usual launch script with **pdb** on the command line. For example:

```
pdb program arguments
```

In launch mode, PDB establishes one connection to each task, and each of these connections consumes one file descriptor. On a Linux system, the number of tasks may exceed the number of file descriptors that a process can open. In this situation, you need to edit the **/etc/security/limits.conf** file to increase the hard limit for the maximum number of file descriptors that can be used per process. For example, root can add the following line for *user1*:

```
user1    hard    nofile    2048
```

After the hard limit in **/etc/security/limits.conf** is increased, *user1* needs to issue **ulimit -n** *hard_limit* before running PDB to set the current limit to the same value as the hard limit.

Using PDB in attach mode

In attach mode, PDB attaches to a program that is already in progress. To use PDB to attach to a running job, use the **-a** flag of the **pdb** command, followed by the process ID of the POE job, and then any client or DISH options you wish to specify. For example:

```
pdb -a 885947 --dish -i 3 --gdb -d source_dir
```

Note that the you must issue this command from the node on which the POE process is running.

Diagnosing problems with PDB

When diagnosing problems with PDB, look at GDB or dbx first. For example, if you see some unexpected behavior while running PDB, you could issue one of the following commands to see if the same behavior exists under **rsh**:

```
rsh host gdb options program
```

Or

```
rsh host dbx options program
```

To obtain more information from PDB, use the **dish -i** *info_level* option.

The following are known issues with PDB:

edit command hangs

- Reason - GDB and dbx launch an external editor to edit source files, which sometimes causes PDB to hang.
- Solution - Do not use the **edit** command. In the event that a user enters the **edit** command, do one of the following:
  - If you are using GDB, press **<Ctrl-c>** to enter the interrupt state, and then type the command **interrupt**.
  - If you are using dbx, press **<Ctrl-c>** to enter the interrupt state, and then enter the command **send :q** twice.

Spurious SIG32 stops the job

- Reason - **pthread_cancel**() calls a program that generates SIG32 when the program is being debugged by GDB.
- Solution - Use the GDB command **handle** *SIG32 nostop*. This command can be put into **.gdbinit** to avoid retyping it for each session.

POE shows: ″ERROR: 0031-652 Error reading STDIN″

- Reason - PDB uses POE to launch DISH agents, and POE is started in the background without STDIN.
- Solution - Ignore this error.

PDB cannot pass an option to GDB, dbx, or POE if there is a space in the option

- Reason - Spaces are treated as word delimitors by the shell script with which PDB is written.
- Solution - Do not use spaces in options.

PDB does not support dbx -r

- Reason - Although **dbx -r** will run a program, when the program ends, dbx will not be launched as the debugger.
- Solution - Do not use **dbx -r** on AIX.

PDB cannot rerun a job

- Reason - LAPI and POE cannot support rerunning a job in the same context.
- Solution - Rerun the job by exiting, and then reentering PDB.

Example: Debugging a program with PDB

In the following example, the user has already set up the execution environment with the same arguments he would use with the **poe** command. Next, he starts PDB in launch mode, specifying the program he wants to debug. In this example, the program to debug runs on Linux and is a bandwidth testcase for MPI, called **mpi_bw**. The launch command looks like this:

```
pdb mpi_bw
```

After issuing the command above, the following output is displayed:

```
PDB -- Parallel Debugger for IBM Parallel Environment

At the prompt, enter any GDB command. Be aware, the tasks
are not yet running. Use debugger's run command to start them.

Enter 'help' for more usage.
(all) break main
0 | Using host libthread_db library "/lib/tls/libthread_db.so.1".
0 | Breakpoint 1 at 0x80497dd: file bw.c, line 291.
1 | Using host libthread_db library "/lib/tls/libthread_db.so.1".
1 | Breakpoint 1 at 0x80497dd: file bw.c, line 291.
```

```
2 | Using host libthread_db library "/lib/tls/libthread_db.so.1".
2 | Breakpoint 1 at 0x80497dd: file bw.c, line 291.
3 | Using host libthread_db library "/lib/tls/libthread_db.so.1".
3 | Breakpoint 1 at 0x80497dd: file bw.c, line 291.
(all) run
0 | Starting program: /u/user/PDB_DEMO/mpi_bw -i 1000 -p
0 | [Thread debugging using libthread_db enabled]
0 | [New Thread 1077082144 (LWP 22912)]
0 | [New Thread 1079184304 (LWP 22918)]
0 | [Switching to Thread 1077082144 (LWP 22912)]
0 |
0 | Breakpoint 1, main (argc=4, argv=0xbff82ad4) at bw.c:291
0 | 291     int num_iter = 10000;
1 | Starting program: /u/user/PDB_DEMO/mpi_bw -i 1000 -p
1 | [Thread debugging using libthread_db enabled]
1 | [New Thread 1077082144 (LWP 25909)]
1 | [New Thread 1079184304 (LWP 25914)]
1 | [Switching to Thread 1077082144 (LWP 25909)]
1 |
1 | Breakpoint 1, main (argc=4, argv=0xbfec7884) at bw.c:291
1 | 291     int num_iter = 10000;
2 | Starting program: /u/user/PDB_DEMO/mpi_bw -i 1000 -p
2 | [Thread debugging using libthread_db enabled]
2 | [New Thread 1077082144 (LWP 22913)]
2 | [New Thread 1079184304 (LWP 22919)]
2 | [Switching to Thread 1077082144 (LWP 22913)]
2 |
2 | Breakpoint 1, main (argc=4, argv=0xbfe4a9e4) at bw.c:291
2 | 291     int num_iter = 10000;
3 | Starting program: /u/user/PDB_DEMO/mpi_bw -i 1000 -p
3 | [Thread debugging using libthread_db enabled]
3 | [New Thread 1077082144 (LWP 25908)]
3 | [New Thread 1079184304 (LWP 25915)]
3 | [Switching to Thread 1077082144 (LWP 25908)]
3 |
3 | Breakpoint 1, main (argc=4, argv=0xbff47714) at bw.c:291
3 | 291     int num_iter = 10000;
(all) print num_iter
0 | $1 = 11127968
1 | $1 = 5565600
2 | $1 = 11127968
3 | $1 = 5565600
(all) next
0 | 292     int mode_begin = PINGPONG, mode_end = BI_STREAMING;
1 | 292     int mode_begin = PINGPONG, mode_end = BI_STREAMING;
2 | 292     int mode_begin = PINGPONG, mode_end = BI_STREAMING;
3 | 292     int mode_begin = PINGPONG, mode_end = BI_STREAMING;
(all) print num_iter
0 | $2 = 10000
1 | $2 = 10000
2 | $2 = 10000
3 | $2 = 10000
(all) group even=0 2
even:  0 2
(all) group odd=1 3
odd:  1 3
(all) on even
(even) continue
0 | Continuing.
2 | Continuing.
(even:int)                                           <-- Control-C pressed
(even:int) interrupt
0 | [New Thread 1196522416 (LWP 22921)]
0 | [New Thread 1198623664 (LWP 22923)]
0 | [New Thread 1209482160 (LWP 22925)]
0 | [New Thread 1211583408 (LWP 22926)]
0 |
0 | Program received signal SIGTRAP, Trace/breakpoint trap.
0 | 0x00a877a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
2 | [New Thread 1196522416 (LWP 22920)]
2 | [New Thread 1198623664 (LWP 22922)]
2 | [New Thread 1209482160 (LWP 22924)]
2 | [New Thread 1211583408 (LWP 22927)]
2 |
2 | Program received signal SIGTRAP, Trace/breakpoint trap.
2 | 0x00a877a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
(even) where
0 | #0  0x00a877a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
0 | #1  0x00b65a41 in ___newselect_nocancel () from /lib/tls/libc.so.6
0 | #2  0x4019f29c in pm_cntl_pipe_select () from /usr/lib/libpoe.so
0 | #3  0x401a0aaa in _udp_info () from /usr/lib/libpoe.so
0 | #4  0x44d66f7f in _get_all_tasks_poe_info () from /usr/lib/liblapiudp.so
0 | #5  0x44d672c9 in _process_empty_ip_addr () from /usr/lib/liblapiudp.so
0 | #6  0x44d683c2 in udp_writepkt () from /usr/lib/liblapiudp.so
0 | #7  0x401d2232 in _send_ready_pkt () from /usr/lib/liblapi.so
0 | #8  0x401d2d79 in _process_epoch_item () from /usr/lib/liblapi.so
0 | #9  0x401d3b31 in _send_processing () from /usr/lib/liblapi.so
0 | #10 0x401c7ef3 in _lapi_dispatcher () from /usr/lib/liblapi.so
0 | #11 0x401c3cbd in _Am_xfer () from /usr/lib/liblapi.so
0 | #12 0x401c4349 in _Dgsp_xfer () from /usr/lib/liblapi.so
0 | #13 0x401c6e88 in PLAPI_Xfer () from /usr/lib/liblapi.so
0 | #14 0x4013b6d0 in mpci_send () from /usr/lib/libmpi_ibm.so
0 | #15 0x4007fb4c in bcast_tree_b () from /usr/lib/libmpi_ibm.so
0 | #16 0x40081e78 in _mpi_bcast () from /usr/lib/libmpi_ibm.so
0 | #17 0x4005e07d in _make_comm () from /usr/lib/libmpi_ibm.so
0 | #18 0x400c0400 in _mpi_init () from /usr/lib/libmpi_ibm.so
0 | #19 0x40121e4c in _css_init () from /usr/lib/libmpi_ibm.so
0 | #20 0x4012294e in _mp_init_msg_passing () from /usr/lib/libmpi_ibm.so
0 | #21 0x400ce8b7 in PMPI_Init () from /usr/lib/libmpi_ibm.so
0 | #22 0x080498bb in main (argc=4, argv=0xbff82ad4) at bw.c:313
2 | #0  0x00a877a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
2 | #1  0x00b65a41 in ___newselect_nocancel () from /lib/tls/libc.so.6
2 | #2  0x4019f29c in pm_cntl_pipe_select () from /usr/lib/libpoe.so
2 | #3  0x401a0aaa in _udp_info () from /usr/lib/libpoe.so
2 | #4  0x44d66f7f in _get_all_tasks_poe_info () from /usr/lib/liblapiudp.so
2 | #5  0x44d672c9 in _process_empty_ip_addr () from /usr/lib/liblapiudp.so
2 | #6  0x44d683c2 in udp_writepkt () from /usr/lib/liblapiudp.so
2 | #7  0x401d2232 in _send_ready_pkt () from /usr/lib/liblapi.so
2 | #8  0x401d2d79 in _process_epoch_item () from /usr/lib/liblapi.so
2 | #9  0x401d3b31 in _send_processing () from /usr/lib/liblapi.so
2 | #10 0x401c7ef3 in _lapi_dispatcher () from /usr/lib/liblapi.so
2 | #11 0x401c3cbd in _Am_xfer () from /usr/lib/liblapi.so
2 | #12 0x401c4349 in _Dgsp_xfer () from /usr/lib/liblapi.so
2 | #13 0x401c6e88 in PLAPI_Xfer () from /usr/lib/liblapi.so
2 | #14 0x4013b6d0 in mpci_send () from /usr/lib/libmpi_ibm.so
2 | #15 0x4007fb4c in bcast_tree_b () from /usr/lib/libmpi_ibm.so
2 | #16 0x40081e78 in _mpi_bcast () from /usr/lib/libmpi_ibm.so
2 | #17 0x4005e07d in _make_comm () from /usr/lib/libmpi_ibm.so
2 | #18 0x400c0400 in _mpi_init () from /usr/lib/libmpi_ibm.so
2 | #19 0x40121e4c in _css_init () from /usr/lib/libmpi_ibm.so
2 | #20 0x4012294e in _mp_init_msg_passing () from /usr/lib/libmpi_ibm.so
2 | #21 0x400ce8b7 in PMPI_Init () from /usr/lib/libmpi_ibm.so
2 | #22 0x080498bb in main (argc=4, argv=0xbfe4a9e4) at bw.c:313
(even) on all
(all) continue
0 | Continuing.
1 | Continuing.
2 | Continuing.
3 | Continuing.
0 |
0 | ------ MPI Ping-pong Bandwidth ------
0 |    1000 x       128 bytes     73.28 us      1.75 MB/s
0 |    1000 x       256 bytes     71.86 us      3.56 MB/s
0 |    1000 x       384 bytes     68.60 us      5.60 MB/s
```

```
0 |    1000 x        512 bytes     79.34 us      6.45 MB/s
(all:int)                                        <-- Control-C pressed
0 |    1000 x        640 bytes     99.96 us      6.40 MB/s
(all:int) leave
0 |    1000 x        768 bytes    134.01 us      5.73 MB/s
0 |    1000 x        896 bytes    132.25 us      6.78 MB/s
0 |    1000 x       1024 bytes    127.32 us      8.04 MB/s
(all:int)                                        <-- Control-C pressed
(all:int) interrupt
0 |
0 | Program received signal SIGTRAP, Trace/breakpoint trap.
0 | 0x00a877a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
1 | [New Thread 1196522416 (LWP 25916)]
1 | [New Thread 1198623664 (LWP 25918)]
1 | [New Thread 1209482160 (LWP 25920)]
1 | [New Thread 1211583408 (LWP 25923)]
1 |
1 | Program received signal SIGTRAP, Trace/breakpoint trap.
1 | 0x005397a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
2 |
2 | Program received signal SIGTRAP, Trace/breakpoint trap.
2 | 0x00a877a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
3 | [New Thread 1196522416 (LWP 25917)]
3 | [New Thread 1198623664 (LWP 25919)]
3 | [New Thread 1209482160 (LWP 25921)]
3 | [New Thread 1211583408 (LWP 25922)]
3 |
3 | Program received signal SIGTRAP, Trace/breakpoint trap.
3 | 0x005397a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
(all) quit
CONSOLE: 2800-801  Connection terminated.
```

Chapter 5. Profiling programs with the prof and gprof commands

You can use the **prof** (AIX only) and **gprof** (AIX and Linux) commands to profile your parallel applications.

The difference between profiling serial and parallel applications with **prof** and **gprof** is that serial applications can be run to generate a single profile data file, while a parallel application can be run to produce many.

Profiling AIX programs with the prof and gprof commands

To profile an AIX program, you request parallel profiling by setting the compile flag to **-p** or **-pg** as you would with serial compilation. The parallel profiling capability of PE creates a monitor output file for each task.

AIX V5.3 TL 5300-05 (and later) allows the profiling output files to have a user-specified name, depending on the setting of **PROF** and **GPROF** environment variables (the **PROF** and **GPROF** environment variables were not supported in AIX 5.2). With AIX V5.3 TL 5300-05 or later, there is additional profiling support for threads and options that affect the type of profiling data that is collected, in addition to other factors that also affect how the profiling output files will be named.

The files are created in the current directory and are named based on the settings of the **PROF** and **GPROF** environment variables, as described below. In all cases *taskid* is a number between 0 and one less than the number of tasks.

- When neither **PROF** nor **GPROF** are set, the default file names are **mon.taskid.out** or **gmon.taskid.out**, respectively.
- When an alternative file name is specified with **PROF**, the parallel profiling output file names are *filename***.taskid.out**.
- When **GPROF** is specified, the resulting output file names are a factor of the keywords specified in the **GPROF** environment variable, as documented by the AIX V5.3 TL 5300-05 (or later) **gprof** command, where the resulting file name will have the *taskid* value appended in the filename prefix string (as defined by the **GPROF filename:** keyword). For example, the following combinations of file names are possible, based on the **GPROF** settings, for parallel profiling output files:
  - For multi file-type: *prefix***-processname-pid.taskid.out**
  - For multithread file-type: *prefix***-processname-pid-Pthread***threaded***.taskid.out**

  The *prefix* default is **gmon**. You can define your own prefix by using the filename parameter of the **GPROF** environment variable. Note that the position where the *taskid* is appended in the file name has changed for parallel profiling output files, beginning with AIX V5.3 TL 5300-05.

In addition, with the added capabilities of the AIX V5.3 TL 5300-05, or later, **GPROF** environment variable, a program compiled with **-pg** potentially produces multiple output files in both the serial and parallel cases, if **profile:thread** is specified as part of **GPROF**. Furthermore, thread profiling capability is only available with profiling output files that are created with AIX V5.3 TL 5300-05, or later. It is strongly suggested that you review the information on the **PROF** and

GPROF environment variables, and the **prof** and **gprof** commands in *AIX Commands Reference* and *AIX General Programming Concepts: Writing and Debugging Programs*.

Following the traditional method of profiling using the AIX operating system, you compile a serial application and run it to produce a single profile data file that you can then process using either the **prof** or **gprof** commands. With a parallel application, you compile and run it to produce a profile data file for each parallel task. You can then process one, some, or all the data files produced using either the **prof** or **gprof** commands.

Table 52 describes how to profile parallel programs. For comparison, the steps involved in profiling a serial program are shown in the left-hand column of the table.

Table 52. Profiling a parallel program, compared to profiling a serial program

To Profile a Serial Program:	To Profile a Parallel Program:
Step 1: Compile the application source code using the **cc** command with either the **-p** or **-pg** flag.	Step 1: Compile the application source code using the command **mpcc_r** (for C programs), **mpCC_r** (for C++ programs), or **mpxlf_r** (for Fortran programs) as described in *IBM Parallel Environment: Operation and Use*. You should use one of the standard profiling compiler options – either **-p** or **-pg** – on the compiler command. For more information on the compiler options **-p** and **-pg**, refer to their use on the **cc** command as described in *AIX Commands Reference* and *AIX General Programming Concepts: Writing and Debugging Programs*.
Step 2: Run the executable program to produce a profile data file. The file name is based on the setting of the **PROF** keyword, in which **mon.out** is the default file name. The file name produced is based on the options that are specified in the **GPROF** keyword, with **gmon** as the default prefix.	Step 2: Before you run the parallel program, set the environment variable **MP_EUILIBPATH=/usr/lpp/ppe.poe/lib/profiled:/usr/lib/profiled:/lib/profiled : /usr/lpp/ppe.poe/lib**. If your message passing library is not in **/usr/lpp/ppe.poe/lib**, substitute your message passing library path. Run the parallel program. When the program ends, it generates a profile data file for each parallel task. The output file for source code that is compiled with the **-p** option is based on the **PROF** keyword setting plus the taskid. In this case, **mon.taskid.out** is the default. The file name produced is based on the options that are specified in the **GPROF** keyword, with **gmon** as the default prefix and the taskid appended. In this case, **gmon.taskid.out** is the default. **Note:** The current directory must be writable from all remote nodes. Otherwise, the profile data files will have to be manually moved to the home node for analysis with **prof** and **gprof**. You can also use the **mcpgath** command to move the files. See *IBM Parallel Environment: Operation and Use* for more about **mcpgath**.

Table 52. Profiling a parallel program, compared to profiling a serial program (continued)

To Profile a Serial Program:	To Profile a Parallel Program:
Step 3: Use either the **prof** or the **gprof** command to process the profile data file. The profile data files are based on the PROF and GPROF environment variable settings.	Step 3: Use either the **prof** or **gprof** command to process the profile data files. The file names are based on the PROF and GPROF environment variable settings. Note that the position in the file name in which taskid is appended has changed for parallel profiling output files on AIX V5.3 TL 5300-05. You can process one, some, or all of the data files created during the run. You must specify the name(s) of the profile data file(s) to read, however, because the **prof** and **gprof** commands read *mon.out* or *gmon.out* by default. On the **prof** command, use the **-m** flag to specify the name(s) of the profile data file(s) it should read. For example, to specify the profile data file for task 0 with the **prof** command: **Assuming the default case, ENTER** **prof -m mon.***0***.out** You can also specify that the **prof** command should take profile data from some or all of the profile data files produced. For example, to specify three different profile data files – the ones associated with tasks 0, 1, and 2 – on the **prof** command: **ENTER** **prof -m mon.***0***.out mon.***1* **.out mon.***2***.out** On the **gprof** command, you simply specify the name(s) of the profile data file(s) it should read on the command line. You must also specify the name of the program on the **gprof** command, but no option flag is needed. For example, to specify the profile data file for task 0 with the **gprof** command: **Assuming the default case, ENTER** **gprof** *program* **gmon.***0***.out** As with the **prof** command, you can also specify that the **gprof** command should take profile data from some or all of the profile data files produced. For example, to specify three different profile data files – the ones associated with tasks 0, 1, and 2 – on the **gprof** command: **ENTER** **gprof** *program* **gmon.***0***.out gmon.***1***.out gmon.***2***.out**

The parallel utility, **mp_profile**( ), may also be used to selectively profile portions of a program. To start profiling, call **mp_profile**(1). To suspend profiling, call **mp_profile**(0). The final profile data set will contain counts and CPU times for the program lines that are delimited by the start and stop calls. In C, the calls are **mpc_profile**(1), and **mpc_profile**(0). By default, profiling is active at the start of the user's executable.

Note: Like the sequential version of **prof**/**gprof**, if more than one profile file is specified, the parallel version of the **prof**/**gprof** command output shows the sum of the profile information in the given profile files. There is no statistical analysis contacted across the multiple profile files.

Profiling Linux programs with the gprof command

To profile a Linux program, you request parallel profiling by setting the compile flag to **-pg** as you would with serial compilation. The parallel profiling capability of PE creates a monitor output file for each task.

The files are created in the current directory, and are identified by the name *gmon.out.taskid*, where *taskid* is a number between 0 and one less than the number of tasks.

Following the traditional method of profiling, you compile a serial application and run it to produce a single profile data file that you can then process using the **gprof** command. With a parallel application, you compile and run it to produce a profile data file for each parallel task. You can then process one, some, or all the data files produced using the **gprof** command. Table 53 describes how to profile parallel programs. For comparison, the steps involved in profiling a serial program are shown in the left-hand column of the table.

Table 53. Profiling a parallel program

To Profile a Serial Program:	To Profile a Parallel Program:
Step 1: Compile the application source code using the **cc** command with the **-pg** flag.	Step 1: Compile the application source code using the command **mpcc** (for C programs), **mpCC** (for C++ programs), or **mpfort** (for Fortran programs) as described in *IBM Parallel Environment: Operation and Use*. You should use the standard profiling compiler option, **-pg**, on the compiler command. For more information on the **-pg** flag, refer to its use on the man page for the compiler you are using.
Step 2: Run the executable program to produce a profile data file. The data file that is produced is named *gmon.out*.	Step 2: Before you run the parallel program, set the environment variable **MP_EUILIBPATH=/opt/ibmhpc/ppe.poe/lib/profiled:/usr/lib/profiled:/lib/profiled : /opt/ibmhpc/ppe.poe/lib**. If your message passing library is not in **/opt/ibmhpc/ppe.poe/lib**, substitute your message passing library path. Run the parallel program. When the program ends, it generates a profile data file for each parallel task. The system gives unique names to the data files by appending each task's identifying number to *gmon.out*. The data files that are produced take the form: **gmon.out.***taskid* **Note:** The current directory must be writable from all remote nodes. Otherwise, the profile data files will have to be manually moved to the home node for analysis with **gprof**. You can also use the **mcpgath** command to move the files. See *IBM Parallel Environment: Operation and Use* for more about **mcpgath**.
Step 3: Use the **gprof** command to process the *gmon.out* profile data file.	Step 3: Use the **gprof** command to process the *gmon.out* profile data files. You can process one, some, or all of the data files created during the run. The **gprof** command reads *gmon.out* by default. On the **gprof** command, you simply specify the name(s) of the profile data file(s) it should read on the command line. You must also specify the name of the program on the **gprof** command. For example, to specify the profile data file for task 0 with the **gprof** command: **ENTER** **gprof** *program* **gmon.out**.*0* You can also specify that the **gprof** command should take profile data from some or all of the profile data files produced. For example, to specify three different profile data files – the ones associated with tasks 0, 1, and 2 – on the **gprof** command: **ENTER** **gprof** *program* **gmon.out.***0* **gmon.out.***1* **gmon.out.***2*

The parallel utility, **mp_profile**( ), may also be used to selectively profile portions of a program. To start profiling, call **mp_profile**(1). To suspend profiling, call **mp_profile**(0). The final profile data set will contain counts and CPU times for the program lines that are delimited by the start and stop calls. In C, the calls are **mpc_profile**(1), and **mpc_profile**(0). By default, profiling is active at the start of the user's executable.

Note: Like the sequential version of **gprof**, if more than one profile file is specified, the parallel version of the **gprof** command output shows the sum of the profile information in the given profile files. There is no statistical analysis contacted across the multiple profile files.

Chapter 6. Parallel Environment commands

PE includes manual pages for all of its user commands.

Each manual page is organized into the sections listed below. The sections always appear in the same order, but some appear in all manual pages while others are optional.

SYNOPSIS
: Includes a diagram that summarizes the command syntax, and provides a brief synopsis of its use and function.

FLAGS
: Lists and describes any required and optional flags for the command.

DESCRIPTION
: Describes the command more fully than the **NAME** and **SYNOPSIS** sections.

ENVIRONMENT VARIABLES
: Lists and describes any applicable environment variables.

EXAMPLES
: Provides examples of ways in which the command is typically used.

FILES
: Lists and describes any files related to the command.

RELATED INFORMATION
: Lists commands, functions, file formats, and special files that are employed by the command, that have a purpose related to the command, or that are otherwise of interest within the context of the command.

cpuset_query

Can be used to verify that memory affinity assignments are performed. **This command applies to PE for Linux only.**

SYNOPSIS

cpuset_query **[pid]**

FLAGS

pid
Optionally, **cpuset_query** can be used to display the memory affinity assignments of a process that is already running, if the command is run outside of poe and the *pid* is specified as input.

DESCRIPTION

cpuset_query is used to verify that memory affinity assignments are performed, as an extension of the POE and LoadLeveler scheduling affinity functions. For more information, see "Managing task affinity on large SMP nodes" on page 57.

The output of **cpuset_query** is written to STDOUT.

EXAMPLES

1. To verify that memory affinity assignments have been performed, use the **cpuset_query** command. For example:

   ```
   poe cpuset_query -task_affinity MCM -procs 4 -pmdlog yes -pmdlog_dir /tmp/user1
   ```

   You will see output similar to this:

   ```
   Total number of MCMs found = 7
   ------number of cpus per MCM = 2
   Total number of CPUs found = 14
   cpuset for process 17355 (1 = in the set, 0 = not included)
   cpu0 = 0
   cpu1 = 0
   cpu2 = 1
   cpu3 = 1
   cpu4 = 0
   cpu5 = 0
   cpu6 = 0
   cpu7 = 0
   cpu8 = 0
   cpu9 = 0
   cpu10 = 0
   cpu11 = 0
   cpu12 = 0
   cpu13 = 0
   cpu14 = 0
   Total number of MCMs found = 7
   ------number of cpus per MCM = 2
   Total number of CPUs found = 14
   cpuset for process 17354 (1 = in the set, 0 = not included)
   cpu0 = 1
   cpu1 = 1
   cpu2 = 0
   cpu3 = 0
   cpu4 = 0
   cpu5 = 0
   ```

```
cpu6 = 0
cpu7 = 0
cpu8 = 0
cpu9 = 0
cpu10 = 0
cpu11 = 0
cpu12 = 0
cpu13 = 0
cpu14 = 0
Total number of MCMs found = 7
------number of cpus per MCM = 2
Total number of CPUs found = 14
cpuset for process 3554 (1 = in the set, 0 = not included)
cpu0 = 1
cpu1 = 1
cpu2 = 0
cpu3 = 0
cpu4 = 0
cpu5 = 0
cpu6 = 0
cpu7 = 0
cpu8 = 0
cpu9 = 0
cpu10 = 0
cpu11 = 0
cpu12 = 0
cpu13 = 0
cpu14 = 0
Total number of MCMs found = 7
------number of cpus per MCM = 2
Total number of CPUs found = 14
cpuset for process 3555 (1 = in the set, 0 = not included)
cpu0 = 0
cpu1 = 0
cpu2 = 1
cpu3 = 1
cpu4 = 0
cpu5 = 0
cpu6 = 0
cpu7 = 0
cpu8 = 0
cpu9 = 0
cpu10 = 0
cpu11 = 0
cpu12 = 0
cpu13 = 0
cpu14 = 0
```

dish

Invokes the Distributed Interactive Shell (dish).

SYNOPSIS

dish *session_number* [**-h**]
[**-i** *info_level*]

The **dish** command invokes DISH (Distributed Interactive Shell), which serves as a control center to multiple distributed copies of a client.

FLAGS

-h
Provides help information.

-i *info_level*
Specifies the level of message reporting. The default is **0**. The following values are valid:

0	Provides no additional information.
1	Provides information, independent of the number of agents and clients.
2	Provides information about connections.
3	Provides information about commands.

session_number
Specifies the existing DISH session to which the console should connect.

DESCRIPTION

The **dish** command invokes DISH (Distributed Interactive Shell), which serves as a control center to multiple distributed copies of a client. Each of these copies is referred to as an instance of the client. For example, Korn Shell can be launched on different nodes and then controlled by DISH. As a result, a command such as **ls** can be broadcast to all shells and executed in parallel to produce the file listing. DISH then gathers the shell output and presents you with the results. DISH can be configured into a distributed shell, a parallel debugger, or some other interactive program, depending on the client you choose to use.

The DISH console communicates with DISH agents on each remote host. The DISH agents relay commands from the DISH console to the client processes, and then report output from the clients back to DISH. The DISH agents connect to DISH through sockets, and are able to spawn multiple client instances. For more information, see “disha” on page 141 and “Using the Distributed Interactive Shell” on page 111.

By default, DISH launches the root agent from the host on which DISH is running. The middle agents are launched either by parent middle agents or the root agent. The leaf agents are launched by the root and middle agents, or by a job launcher such as POE.

By default, **rsh** (Remote Shell) is used for launching agents, but if you want to use a different method, you can use the **-r** flag with the **disha** command. For example, if you wanted to use **ssh** (Secure Shell) instead of **rsh**, you would specify **disha -r** *ssh*.

Note that if you have a large number of client sessions, you could experience serious bottlenecks when they are launched. To avoid situations such as this, DISH uses a tree algorithm to cause agents to launch other agents. The root agent generates a tree structure from the instance file and the fanout factor. The fanout factor can be specified with the **-f** option of the **disha** command. The root agent uses **rsh** or **ssh** to launch the next level child agents, and the child agents connect back to the root agent after startup. In turn, the child agents then launch their own child agents.

In addition to DISH, you can also use other job launching mechanisms, for example, POE, to launch DISH agents. Before you do this, you must first tell DISH not to launch leaf agents. You do this with the **disha -n** flag. Next, you need to tell the DISH agents where DISH is running and also supply them with an agent ID for connection. In general, a task ID is available from the job launcher, which can be used as the agent ID.

DISH allows you to manage clients collectively through the use of groups. The **dish** command provides subcommands and options for gathering clients into groups, and also for managing those groups. For example, if a user issues a client command, that command can be broadcast to all members of the group. You can create your own groups or you can use one of the predefined groups supplied with DISH. For more information on using groups with DISH, see "group subcommand (of the dish command)" on page 137.

With DISH, you can create groups of clients, and then use DISH to manage those groups. For example, when a user issues a client command, that command can then be broadcast to all members of the group. At any given time, you can have multiple groups of clients. The group subcommand allows you to easily switch DISH's focus from one group of clients to another. DISH also includes predefined groups. For more information about the group subcommand, see

INSTANCE FILE:

When you set up a DISH session, you must create an *instance* file to, at a minimum, specify the hosts on which the client instances will run. The instance file defines instance-specific information only. For example, if you wanted to launch DISH with a debugger, you would include the process IDs that are required by DISH in order to attach to a running job. Here is what that instance file might look like:

```
host pid
########################
wcsvr03.rd.acme.com 20396
wcsvr04.rd.acme.com 20379
wcsvr03.rd.acme.com 20397
wcsvr04.rd.acme.com 20380
```

The format of the instance file is similar to a plaintext database, where fields are defined, followed by records consisting of the values for those fields. The first line of the instance file should contain fields for which you will be providing values (*host* and *pid*, in the example above). The values should be entered below the corresponding fields, line-by-line. Note that the field names are case-sensitive.

For every DISH session, you must specify at least the host in the instance file. Also note that you can place the **host** variable anywhere in the list of fields. More variables can be defined in the instance file to provide other information required by the client instances, and all variables can be referred through variable substitution. For more information see "Using variable substitution" on page 114.

The names you add to the instance file must not conflict with existing variable names. For example, *pid* could also be called **process_id** or **procID**.

CONFIGURATION FILE:

In addition to the instance file, DISH also requires a run-time configuration file to provide information about the client and control the behavior of DISH. It allows DISH to support different client programs without knowing what functions they are performing. Unlike the instance file, the configuration file defines session-wide information, rather than information about specific client instances (every instance sees the same value).

In the configuration file, you define a list of variables followed by their values (**variable** = *value*). Both the variables and their values are case-sensitive.

DISH recognizes the following variables:
- **client** - defines the command line to launch the client.
- **client_prompt** - defines the prompt of the client. The prompt must not change during a DISH session.
- **interrupt_command** - defines the command sequence to be used to interrupt a running client process. When a running client process is successfully interrupted, the prompt should appear again. This command sequence is issued against each client in the current group.
- **key_enter** - for convenience, **key_enter** defines the command to execute when the <Enter> key has been pressed on any empty line.

Clients may require other information to be defined in the instance and configuration files in order to function properly. For example, when a debugger is started, it may require the location of a target program, or it may need to know a process ID to which it should attach. But DISH knows nothing about the functions of the clients, and only recognizes its own specific fields. As a result, you need to add any non-DISH variable values to the instance/configuration files, and then specify those variables using variable substitution. The values that you define in the instance/configuration file are used as the substitution values. With variable substitution, the values to be substituted are prefixed by % (percent sign).

DISH recognizes two kinds of variables; instance-specific variables and session-wide variables. An instance-specific variable has a set of values, one for each instance. A session-wide variable has only one value, which is used for all instances.

The following example shows the **client** variable in the configuration file. In this example, GDB (GNU debugger) is the client. The **-quiet** flag suppresses the GDB prolog. The rest of the line asks DISH to substitute *%exe* with the value of the **exe** variable and *%pid* with the value of the **pid** variable.

```
client = gdb -quiet  %exe %pid
```

The **exe** variable is session-wide for SIMD parallel programs, but can be instance-specific for MIMD parallel programs also. The **pid** variable is instance-specific.

VARIABLE SUBSTITUTION:

After you define non-DISH values in the instance and configuration files, variable substitution can be used to define values when you enter commands. For example, you could issue the **attach** *%pid* command to have GDB attach to running processes. The variable *%pid* is substituted by the value you defined for **pid** in the instance file before the **attach** command is passed to GDB.

To summarize, any variable that starts with a % will be substituted. Different client instances get different substitutions if the variable is instance-specific.

Reserved variables are recognized by DISH but their values must be defined by you. The values for *predefined variables* are predefined by DISH, which allows you to reference them freely.

The reserved DISH variables are:
- **host** - is instance-specific. Contains the name of the host on which the client runs.
- **client** - is session-wide. Defines the command line to launch the client.
- **client_prompt** - is session-wide. defines the prompt of the client.
- **interrupt_command** - is session-wide. Defines the command sequence to be used to interrupt a running client process.
- **key_enter** - is session-wide. Defines the command to execute when the <Enter> key has been pressed on any empty line.

The predefined DISH variables are:
- **id** - is instance-specific. The i-th instance in the instance file gets i minus 1 as its **id**.
- **client_pid** - is instance-specific. Is the process ID of an instance.
- **child_pid** - is instance-specific. Contains the child process IDs of the instance. Multiple process IDs are separated by spaces. **child_pid** is dynamic because child processes may come and go during the life of an instance.

DISH STATES:

DISH can be in three different states, in terms of readiness for input. The states are:
- Ready state
  - All clients in the current group have prompted for input.
  - DISH also prompts for input (for example, by showing *(all)*).
  - Client commands are available. DISH commands, except those for interrupt state only, are also available.
- Execution state
  - At least one client in the current group has not prompted for input.
  - DISH does not prompt for input, but shows client output, if any.
  - No commands are available, but you can press <Ctrl-C> or send a SIGINT to DISH to get into interrupt state.
- Interrupt state
  - At least one client in the current group has not prompted for input but you pressed <Ctrl-C> or sent a SIGINT.
  - DISH prompts for input and indicates interrupt state (for example, by showing *(all:int)*). Client output is not displayed.
  - All DISH commands are available but not client commands. Refer to the **interrupt** and **leave** commands, which are available only in interrupt state.

With DISH, there are two types of commands; DISH commands and client commands. DISH commands provide the capability of managing clients in groups, while client commands are implemented by the client. Any command that DISH does not understand is passed to the clients. The availability of commands depends on the state of DISH.

The **dish** command provides the following subcommands that are available for controlling a session. They are:

- **group** - for defining groups of clients
- **on** - for designating a particular group of clients as the current group
- **interrupt** - for interrupting execution of the clients that belong to the current group
- **leave** - for causing the current group to leave the interrupt state
- **kquit** - for killing all running clients and quitting **dish**
- **help** - for getting help information on any of the **dish** subcommands, or on predefined **dish** topics.
- **exit** - for quitting the DISH console.
- **send** - for sending secondary prompt messages to the current group
- **toggle** - for setting toggle mode on and off.
- **match** - for applying filter rules to command output

For more information on the **dish** subcommands see "Subcommands of the dish command" on page 137.

Refer to the client's documentation for information about client commands. To indicate that a command is a client command, you can prefix it with a colon (:). For example, issuing **:help** will bypass DISH and output the client's help instead. Client commands are generally broadcast to all members in the current group. A help command with a client command or topic that is not preceded by a colon (:) is sent to only one client.

EXAMPLES

To configure DISH into a distributed shell, the configuration file would look similar to this:

```
client = ksh -i
client_prompt = $      # change '$' to '#' for root
interrupt_command = kill -INT %child_pid
key_enter = echo `hostname`:`pwd`
```

See the **pdb** man page and the **/usr/bin/pdb** shell script for information about building a parallel debugger on top of DISH.

RELATED INFORMATION

Commands: **disha**(1), **pdb**(1), **poe**(1), **rsh**(1), **ssh**(1)

Subcommands of the dish command

There are a number of subcommands that can be used with the **dish** command. They are: **group**, **help**, **interrupt**, **kquit**, **leave**, and **on**.

When using **dish** subcommands, note the following:
- Commands that start with a colon (:) are passed directly to a client.
- **dish** subcommands can be abbreviated. For example, the **group** subcommand can be entered as **g**, **gr**, **gro**, and so on.

group subcommand (of the dish command)

The **group** subcommand allows you to define a group of clients. It also lets you add and delete clients from existing groups.

group
[*name*]
[*name* = *clients*]
[*name* + *clients*]
[*name* - *clients*]

In the synopsis above, *name* refers to the name of a group. *clients* refers to a list of clients, separated by a space. You can also specify group names in the list of clients. For example:

```
group a = 0:2 4
```

Results in a group that consists of clients 0, 1, 2, and 4.

```
group b = 3 a
```

Results in a group that consists of clients 0, 1, 2, 3, and 4.

```
group a + b 5:7
```

Results in a group that consists of clients 0, 1, 2, 3, 4, 5, 6, and 7.

```
group b - 0 4
```

Results in a group that consists of clients 1, 2, and 3.

DISH also provides the following predefined groups:
- **all** – all client instances
- **0**, **1**, **2**, **...**, *number_of_clients* **-1** – individual client instances
- **ready** – all client instances that are ready for input

help subcommand (of the dish command)

The **help** subcommand provides help information about DISH topics or commands.

help [*command* | *topic*]

To use the **help** subcommand, type **help**, followed by the name of the topic or the name of a command. For example, if you wanted help information on the **dish group** subcommand, you would enter the following:

```
help group
```

Or if you wanted help information on DISH restrictions, you would enter the following:

```
help restrictions
```

The **help** subcommand provides help on the following other **dish** subcommands:
- **group**
- **on**
- **interrupt**
- **leave**
- **kquit**
- **toggle**
- **match**
- **send**
- **exit**

The **dish help** subcommand provides help on the following topics:
- **general** - Provides basic information about dish.
- **states** - Provides information about the current state of DISH and DISH's clients.
- **restrictions** - Lists the client commands and features you should avoid.
- **editing** - Provides information on command editing.

To get an overview of DISH commands and topics, issue the **help** subcommand with no arguments.

interrupt subcommand (of the dish command)

The **interrupt** subcommand stops the execution of a client in the current group.

interrupt

By default, DISH sends a SIGINT to the running client process in order to stop it. However, you can use the **interrupt_command** variable, of the configuration file, to define a sequence of UNIX commands that should be used to stop clients instead. When you set the **interrupt_command** variable, it applies to all clients in the session.

Note that a group can be designated as the current group by using the **dish on** subcommand.

kquit subcommand (of the dish command)

kquit kills all running client processes by sending a SIGKILL, and then quitting DISH. **kquit** sends the SIGKILL to the clients' child processes first, and then to the clients themselves.

kquit

The **kquit** subcommand should only be used in cases where the client's **quit** or **exit** command does not work.

leave subcommand (of the dish command)

The **leave** subcommand is used to end the interrupt state for the current group.

leave

This command can only be used when the current group is in the interrupt state.

on subcommand (of the dish command)

The **on** subcommand is used to specify that a group should become the current group.

on *group*

Client commands are broadcast to all the members of the current group.

exit subcommand (of the dish command)

exit quits the current DISH console.

exit

The **exit** subcommand is used to quit the DISH console. This applies only to the current DISH console. If no other consoles are connected when **exit** is issued, the entire debugging session ends.

send subcommand (of the dish command)

send sends secondary prompt messages to the group.

send *message*

The **send** subcommand is used to send secondary prompt messages to the current group. This subcommand can be used only when the DISH console is in interrupt state.

toggle subcommand (of the dish command)

toggle sets toggle mode on and off.

toggle [**broadcast**] [**color** *name*] [**bcolor** *name*] [**interval** *second*]

The **toggle** subcommand can be used when multiple consoles are attached to the same debugging session, and the DISH console is in the running state. When the toggle is enabled, one DISH console can display messages from all other connected consoles.

toggle [broadcast]
: Turns toggle mode on and off. When toggle mode is on, message output from other consoles is displayed. The broadcast suboption is available in normal state only.

toggle [color | bcolor *name***]**
: Sets the foreground color of messages. The default is green. The **color** suboption is used to set the color of messages on the current console. The **bcolor** suboption is used to set the color of messages from other consoles.

name can be set to any of the following values:
- black

- red
- green
- yellow
- darkblue
- purple
- blue
- white
- gray

Examples:
- `toggle color blue`
- `toggle bcolor yellow`

toggle [interval *second***]**
Sets the message flush interval. The default is 5 seconds. *second* can be any integer from 0 to 30.

Example:
- `toggle interval 3`

match subcommand (of the dish command)

match filters client command output.

match *'regular_expression'*

The **match** subcommand can be used to apply filter rules to client command output using regular expressions. The regular expression must be surrounded by quotation marks.

match Shows the last client command output.

match '*regular_expression***'**
Shows the lines that contain the specified regular expression in the last client command.

match '*regular_expression***' | match '***regular_expression***' ...**
Shows the lines that contain the specified regular expressions in the last client command, in cascade style.

client_command **match '***regular_expression***'**
Shows the lines, from the command output, that contain the specified regular expression

client_command **match '***regular_expression***' | match '***regular_expression***'**
Shows the lines, from the command output, that contain the specified regular expressions, in cascade style. The **match** suboption is available in normal state only.

Examples:
- `match 'Hello'`
- `match 'Hello' | match "World"`
- `list | match 'Hello'`
- `list | match 'Hello' | match 'World'`

disha

Invokes a remote DISH agent.

SYNOPSIS

disha [**-e**] [**-h**][**-n**]
[**-a** *agent_id*]
[**-c** *config_file*]
[**-f** *fanout_factor*]
[**-g** *variable=value*]
[**-H** *server_host*]
[**-i** *info_level*]
[**-I** *instance_file*]
[**-k** *session_key*]
[**-p** *server_port*]
[**-r** *remote_shell*]
[**-s** *session_number*]
[**-t** *task_id*]

The **disha** command invokes the remote agent for DISH.

FLAGS

-a *agent_ID*
Specifies the ID of the DISH agent.

It must be either an integer or the name of an environment variable that specifies the root, middle, or leaf agent ID at run time. A value of 0 indicates that this is the root agent. Each agent must have a unique agent ID.

-c *config_file*
Specifies the location of the configuration file to be used for a DISH session. The default location is your home directory (~/**.dish**).

-e Indicates that this agent is a leaf agent, and that it is launched by an external job launcher.

-f *fanout_factor*
Specifies the fanout factor for launching DISH agents. The default value is **INT_MAX**.

-h Provides help information.

-H *server_host*
Specifies the host on which the agent's parent is running.

-i *info_level*
Specifies the level of message reporting. The default is **0**. The following values are valid:

0	Provides no additional information.
1	Provides information, independent of the number of agents and clients.
2	Provides information about connections.
3	Provides information about commands.

-I *instance_file*
Specifies the file that contains the list of hosts on which to run clients. The

default location is your home directory (~/**.dish**). For more information about the instance file, see "Creating an instance file" on page 112.

-k *session_key*
Specifies the session key for connection validation. The default is **1**.

-p *server_port*
Specifies the port to which to the parent agent is listening for incoming connections.

-r *remote_shell*
Specifies the remote shell to use for launching clients. The default is **rsh**.

-a *session_number*
Specifies the session number from this DISH session. The default is **1**.

-t *task_id*
Specifies the ID of the client instance of a leaf DISH agent, if this agent has been launched by an external job launcher.

DESCRIPTION

The **disha** command invokes a DISH remote agent. For more information on DISH and the role of the DISH agent, see "Using the Distributed Interactive Shell" on page 111.

RELATED INFORMATION

Commands: **dish**(1)

mcp

Allows you to propagate a copy of a file to multiple nodes.

SYNOPSIS

mcp *infile* [*outfile*] [*POE options*]

In the previous command synopsis, the *infile* is the name of the file to be copied. You can copy to a new name by specifying an *outfile*. If you do not provide the outfile name, the file will be placed in its current directory on each node. The outfile can be either an explicit output file name or a directory name. When a directory is specified, the file is copied with the same name to that directory.

DESCRIPTION

The **mcp** command allows you to propagate a copy of a file to multiple nodes. The file must initially reside (or be NFS-mounted) on at least one node.

mcp is a POE program and, therefore, all POE options are available. You can set POE options with either command line flags or environment variables. The number of nodes to copy the file to (**-procs**), and the message passing protocol used to copy the file (**-euilib**) are the POE options of most interest. The input file must be readable from the node assigned to task 0.

Note: A POE job loads faster if a copy of the job resides on each node. For this reason, it is suggested that you use **mcp** to copy your executable to a file system such as /tmp, which resides on each node.

Return codes are:

129
incorrect usage

130
error opening input file

131
error opening *to* file on originating node

132
error writing data to *to* file on originating node

133
no room on remote node's file system

134
error opening file on remote node

135
error writing data on remote node

136
error renaming temp file to file name

137
input file is empty

138
invalid block size

139
: error allocating storage

ENVIRONMENT VARIABLES

MP_BLKSIZE
: Sets the block size used for copying the data. This can be a value between 1 and 8,000,000 (8 megabytes). The default is 100,000 (100K).

EXAMPLES

1. To copy a file from your current directory to the current directory for 16 tasks, using the User Space protocol, enter:

   ```
   mcp filename -procs 16 -euilib us
   ```
2. To copy a filename from your current directory to the /tmp directory for 16 tasks, using IP, enter:

   ```
   mcp filename /tmp -procs 16 -euilib ip
   ```
3. To copy a file from your current directory to a different filename for 16 tasks, enter:

   ```
   mcp filename /tmp/newfilename -procs 16
   ```

RELATED INFORMATION

Commands: **rcp**(1)

mcpgath

Takes files from each task of tasks 0 to n-1 and copies them back in sequence to task 0.

SYNOPSIS

mcpgath [**-ai**] *source ... destination* [*POE options*]

Source is one of the following:

- one or more existing file names - files will be copied with the same names to the destination directory on task 0. Each file name specified must exist on all tasks involved in the copy.
- a directory name - all files in that directory on each task are copied with the same names to the destination directory on task 0.
- an expansion of file names, using wildcards - files are copied with the same names to the destination directory. All wildcarded input strings must be enclosed in double quotes.

Destination is an existing destination directory name to where the data will be copied. The destination directory must be the last item specified before any POE flags.

FLAGS

-a An optional flag that appends the task number to the end of the file name when it is copied to task 0. This is for task identification purposes, to know where the data came from. The **-a** and **-i** flags can be combined to check for existing files appended with the task number.

-i An optional flag that checks for duplicate or existing files of the same name, and does not replace any existing file found. Instead, issues an error message and continues with the remaining files to be copied. The **-a** and **-i** flags can be combined to check for existing files appended with the task number.

See Chapter 2, "Executing parallel programs," on page 11 for information on POE options.

DESCRIPTION

The **mcpgath** function determines the list of files to be gathered on each task. This function also resolves the source file, destination directory, and path names with any meta characters, wildcard expansions, and so on, to come up with valid file names. Enclose wildcards in double quotes, otherwise they will be expanded locally on the task from where the command is issued, which may not produce the intended file name resolution.

mcpgath is a POE program and, therefore, all POE options are available. You can set POE options with either command line flags or environment variables. The number of nodes to copy the file to (**-procs**), and the message passing protocol used to copy the file (**-euilib**) are the POE options of most interest.

Return codes are:

129
invalid number of arguments specified

130
invalid option flag specified

131
unable to resolve input file name(s)

132
could not open input file for read

133
no room on destination node's file system

134
error opening file output file

135
error creating output file

136
error writing to output file

137
MPI_Send of data failed

138
final **MPI_Send** failed

139
MPI_Recv failed

140
invalid block size

141
error allocating storage

142
total number of tasks must be greater than one

ENVIRONMENT VARIABLES

MP_BLKSIZE
Sets the block size used for copying the data. This can be a value between 1 and 8,000,000 (8 megabytes). The default is 100,000 (100K).

EXAMPLES

1. You can copy a single file from all tasks into the destination directory. For example, enter:

   ```
   mcpgath -a hello_world /tmp -procs 4
   ```

   This will copy the file *hello_world* (assuming it is a file and not a directory) from tasks 0 through 3 as to task 0:

   ```
   From task 0:  /tmp/hello_world.0

   From task 1:  /tmp/hello_world.1

   From task 2:  /tmp/hello_world.2

   From task 3:  /tmp/hello_world.3
   ```
2. You can specify any number of files as source files. The destination directory must be the last item specified before any POE flags. For example:

```
mcpgath -a file1.a file2.a file3.a file4.a file5.a /tmp -procs 4
```

will take *file1.a* through *file5.a* from the local directory on each task and copy them back to task 0. All files specified must exist on all tasks involved. The file distribution will be as follows:

```
From Task 0: /tmp/file1.a.0

From Task 1: /tmp/file1.a.1

From Task 2: /tmp/file1.a.2

From Task 3: /tmp/file1.a.3

From Task 0: /tmp/file2.a.0

From Task 1: /tmp/file2.a.1

From Task 2: /tmp/file2.a.2

From Task 3: /tmp/file2.a.3

From Task 0: /tmp/file3.a.0

From Task 1: /tmp/file3.a.1

From Task 2: /tmp/file3.a.2

From Task 3: /tmp/file3.a.3

From Task 0: /tmp/file4.a.0

From Task 1: /tmp/file4.a.1

From Task 2: /tmp/file4.a.2

From Task 3: /tmp/file4.a.3

From Task 0: /tmp/file5.a.0

From Task 1: /tmp/file5.a.1

From Task 2: /tmp/file5.a.2

From Task 3: /tmp/file5.a.3
```

3. You can specify wildcard values to expand into a list of files to be gathered. For this example, assume the following distribution of files before calling **mcpgath**:

```
Task 0 contains file1.a and file2.a

Task 1 contains file1.a only

Task 2 contains file1.a, file2.a, and file3.a

Task 3 contains file4.a, file5.a, and file6.a
```

Enter:

```
mcpgath -a "file*.a" /tmp -procs 4
```

This will pass the wildcard expansion to each task, which will resolve into the list of locally existing files to be copied. This results in the following distribution of files on task 0:

```
From Task 0: /tmp/file1.a.0

From Task 0: /tmp/file2.a.0

From Task 1: /tmp/file1.a.1

From Task 2: /tmp/file1.a.2

From Task 2: /tmp/file2.a.2

From Task 2: /tmp/file3.a.2

From Task 3: /tmp/file4.a.3

From Task 3: /tmp/file5.a.3

From Task 3: /tmp/file6.a.3
```

4. You can specify a directory name as the source, from which the files to be gathered are found. For this example, assume the following distribution of files before calling **mcpgath**:

   ```
   Task 0 /test contains file1.a and file2.a

   Task 1 /test contains file1.a only

   Task 2 /test contains file1.a and file3.a

   Task 3 /test contains file2.a, file4.a, and file5.a
   ```

   Enter:

   ```
   mcpgath -a /test /tmp -procs 4
   ```

   This results in the following file distribution:

   ```
   From Task 0: /tmp/file1.a.0

   From Task 0: /tmp/file2.a.0

   From Task 1: /tmp/file1.a.1

   From Task 2: /tmp/file1.a.2

   From Task 2: /tmp/file3.a.2

   From Task 3: /tmp/file2.a.3

   From Task 3: /tmp/file4.a.3

   From Task 3: /tmp/file5.a.3
   ```

mcpscat

Takes a number of files from task 0 and scatters them in sequence to all tasks, in a round-robin order.

SYNOPSIS

mcpscat [**-f**] [**-i**] *source ...*
destination
[*POE options*]

Source can be one of the following:
- a single file name - file is copied to all tasks
- a single file name that contains a list of file names (**-f** option)
- two or more file names - files will be distributed in a round-robin order to the tasks
- an expansion of file names, using wildcards - files will be distributed in a round-robin order to the tasks
- a directory name - all files in that directory are copied in a round-robin order to the tasks.

Destination is an existing destination directory name to where the data will be copied.

FLAGS

-f Is an optional flag that indicates that the first file contains the names of the source files that are to be scattered. Each file name, in the file, must be specified on a separate line. No wildcards are supported when this option is used. Directory names are not supported in the file either. When this option is used, the **mcpscat** parameters should consist of a single source file name (for the list of files) and a destination directory. The files will then be scattered just as if they had all been specified on the command line in the same order as they are listed in the file.

-i Checks for duplicate or existing files of the same name, and does not replace any existing file found. Instead, issues an error message and continues with the remaining files to be copied. Without this flag, the default action is to replace any existing files with the source file.

See Chapter 2, "Executing parallel programs," on page 11 for information on POE options.

DESCRIPTION

The **mcpscat** function determines the order in which to distribute the files, using a round-robin method, according to the list of nodes and number of tasks. Files are sent in a one-to-one correspondence to the nodes in the list of tasks. If the number of files specified is greater than the number of nodes, the remaining files are sent in another round through the list of nodes. Enclose wildcards in double quotes, otherwise they will be expanded locally on the task from where the command is issued, which may not produce the intended file name resolution.

mcpscat is a POE program and, therefore, all POE options are available. You can set POE options with either command line flags or environment variables. The number of nodes to copy the file to (**-procs**), and the message passing protocol used to copy the file (**-euilib**) are the POE options of most interest.

Return codes are:

129
invalid number of arguments specified

130
invalid option flag specified

131
unable to resolve input file name(s)

132
could not open input file for read

133
no room on destination node's file system

134
error opening file output file

135
error creating output file

136
MPI_Send of data failed

137
final **MPI_Send** failed

138
MPI_Recv failed

139
failed opening temporary file

140
failed writing temporary file

141
error renaming temp file to filename

142
input file is empty (zero byte file size)

143
invalid block size

144
error allocating storage

145
number of tasks and files do not match

146
not enough memory for list of file names

ENVIRONMENT VARIABLES

MP_BLKSIZE
Sets the block size used for copying the data. This can be a value between 1 and 8,000,000 (8 megabytes). The default is 100,000 (100K).

EXAMPLES

1. You can copy a single file to all tasks into the destination directory. For example, enter:

   ```
   mcpscat filename /tmp -procs 4
   ```

   This will take the file and distribute it to tasks 0 through 3 as */tmp/filename*.
2. You can specify any number of files as source files. The destination directory must be the last item specified before any POE flags. For example:

   ```
   mcpscat file1.a file2.a file3.a file4.a file5.a /tmp -procs 4
   ```

   will take *file1.a* through *file5.a* from the local directory and copy them in a round-robin order to tasks 0 through 3 into */tmp*. The file distribution will be as follows:

   ```
   Task 0: /tmp/file1.a

   Task 1: /tmp/file2.a

   Task 2: /tmp/file3.a

   Task 3: /tmp/file4.a

   Task 0: /tmp/file5.a
   ```
3. You can specify the source files to copy in a file. For example:

   ```
   mcpscat -f file.list /tmp -procs 4
   ```

   will produce the same results as the previous example if as *file.list* contains five lines with the file names *file1.a* through *file5.a* in it.
4. You can specify wildcard values to expand into a list of files to be scattered. Enter:

   ```
   mcpscat "file*.a" /tmp -procs 4
   ```

   Assuming Task 0 contains *file1.a*, *file2.a*, *file3.a*, *file4.a*, and *file5.a* in its home directory, this will result in a similar distribution as in the previous example.
5. You can specify a directory name as the source, from which the files to be scattered are found. Assuming */test* contains *myfile.a*, *myfile.b*, *myfile.c*, *myfile.d*, *myfile.f*, and *myfile.g* on Task 0, enter:

   ```
   mcpscat /test /tmp -procs 4
   ```

   This results in the following file distribution:

   ```
   Task 0: /tmp/myfile.a

   Task 1: /tmp/myfile.b

   Task 2: /tmp/myfile.c

   Task 3: /tmp/myfile.d

   Task 0: /tmp/myfile.f

   Task 1: /tmp/myfile.g
   ```

mpamddir

Echoes an amd-mountable directory name.

SYNOPSIS

mpamddir

or, if you're using the Parallel Environment:

export *MP_REMOTEDIR=mpamddir*

This script determines whether or not the current directory is a mounted file system. If it is, it looks to see if it appears in the amd maps, and constructs a name for the directory that is known to amd. You can modify this script, or create additional ones that apply to your installation.

By default, POE uses the Korn shell **pwd** command to obtain the name of the current directory to pass to the remote nodes for execution. This works for C shell users if the current directory is:
- The home directory
- Not mounted by amd, the AutoMount Daemon.

If this is not the case, (for example, if the user's current directory is a subdirectory of the home directory), then you can supply your own script for providing the name of the current directory on the remote nodes.

To use **mpamddir** as the script for providing the name, export the environment variable **MP_REMOTEDIR**, and set it to **mpamddir**.

RELATED INFORMATION

Commands: **ksh**(1), **poe**(1), **csh**(1)

mpcc

Invokes a C compiler to compile C programs that use MPI. **This command applies to PE for Linux only.**

SYNOPSIS

mpcc [*cc_flags*]... *program.c*

The **mpcc** shell script compiles C programs while linking in the Partition Manager, the Message Passing Interface (MPI), and Low-level Applications Programming Interface (LAPI).

You can specify a C compiler with the **MP_COMPILER** environment variable. If the specified compiler is one of the C compilers supported by the IBM C **Set++ vac.cmp** package, or is the GNU GCC compiler, it is invoked using its default install path. It is assumed that any other compiler is reachable via the user's regular search path.

If **MP_COMPILER** is not defined, the **mpcc** shell script first searches for the xlc_r compiler of the IBM C **Set++ vac.cmp** package. If the IBM C **Set++ vac.cmp** package is not installed, it looks for the GNU GCC compiler.

FLAGS

The **mpcc** shell script passes most flags directly to the compiler. Some flags are interrupted by the **mpcc** shell script. These are:

-h prints a help message.

-v is passed to the compiler and causes a *verbose* output listing of the shell script.

-m32 | -q32
enables compiling of 32-bit applications (default). The **q** flag is used by the IBM compiler and the **m** flag is used by the GNU GCC compiler. The **mpcc** script accepts them interchangeably and passes the right flag to the right compiler.

-m64 | -q64
enables compiling of 64-bit applications. The **q** flag is used by the IBM compiler and the **m** flag is used by the GNU GCC compiler. The **mpcc** script accepts them interchangeably and passes the right flag to the right compiler.

-compiler
specifies the name of the C compiler to use. The default value is **xlc_r**, if the IBM C **Set++ vac.cmp** package is installed. If it is not installed, the default value becomes **gcc** (the GNU GCC compiler).

You can use the **-compiler** flag to specify a compiler other than **xlc_r**. For example, you may want to use the GNU GCC compiler, even if the IBM C **Set++ vac.cmp** package is installed. Or, you may want to specify one of the other compilers that are contained in the IBM C **Set++ vac.cmp** package.

To specify a third-party compiler, you must either specify its full path or add that compiler to a directory in your search path (for example, **/usr/bin**).

Other commonly used C compiler flags are:

-g Produces an object file with symbol table references. This object file is needed for debugging with the gdb debugger.

-o names the executable.

-l (lowercase L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the *Context* section of this manual page.

-I (uppercase i)
names directories for additional includes. The directory **/opt/ibmhpc/ppe.poe/include** or the appropriate subdirectory is included automatically. Command line or makefile hard-coding of include paths for PE header files should normally be avoided. Such specifications take precedence over the directory selected by the script and may cause incorrect code to be generated.

-pg
enables profiling with the **gprof** command. For more information, see the information on profiling programs in *IBM Parallel Environment: Operation and Use*.

DESCRIPTION

The **mpcc** shell script invokes a C compiler to compile C programs that use MPI. In addition, the Partition Manager and data communication interfaces are automatically linked in. The script creates an executable that dynamically binds with the communication subsystem libraries.

You can specify a C compiler using the **MP_COMPILER** environment variable.

Most C compiler flags are passed directly from **mpcc** to the compiler. The communication subsystem library implementation is dynamically linked when you invoke the executable using the **poe** command.

ENVIRONMENT VARIABLES

MP_COMPILER
Specifies the name of the C compiler to use. The default value is **xlc_r**, if the IBM C **Set++ vac.cmp** package is installed. If it is not installed, the default value becomes **gcc** (the GNU GCC compiler).

You can use **MP_COMPILER** to specify a compiler other than **xlc_r**. For example, you may want to use the GNU GCC compiler, even if the IBM C **Set++ vac.cmp** package is installed. Or, you may want to specify one of the other compilers that are contained in the IBM C **Set++ vac.cmp** package.

To specify a third-party compiler, you must either specify its full path or add that compiler to a directory in your search path (for example, **/usr/bin**).

MP_PREFIX
Sets an alternate path to the default IBM PE library and include file path. If not set or NULL, the default library path is **/usr/lib**, and the default include file path is **/opt/ibmhpc/ppe.poe/include**. If this environment variable is set, all library search paths provided by **mpcc** are prefixed by **$MP_PREFIX/lib**, and all include file search paths are prefixed by **$MP_PREFIX/include**.

OBJECT_MODE
Setting this variable to 64 causes the 64-bit libraries to be linked to the

executable, as if the **-q64** option had been set. If you do not set **OBJECT_MODE**, or if you set it to anything other than *64* , the 32-bit libraries will be linked to the executable.

EXAMPLES

To compile a C program, enter:

```
mpcc program.c -o program
```

FILES

When you compile a program using **mpcc**, the following libraries are automatically selected.

- Message Passing Interface, collective communication routines:
  - **/usr/lib/libmpi_ibm.so**
  - **/usr/lib/libpoe.so**
  - **/usr/lib64/libmpi_ibm.so**
  - **/usr/lib64/libpoe.so**
- IBM Low-Level Applications Programming Interface routines:
  - **/usr/lib/liblapi.so**
  - **/usr/lib64/liblapi.so**

RELATED INFORMATION

Commands: **mpCC**(1), **mpfort**(1)

mpcc_r

Invokes a shell script to compile C programs that use MPI. **This command applies to PE for AIX only.**

SYNOPSIS

mpcc_r [*cc_flags*]... *program.c*

The **mpcc_r** shell script compiles C programs while linking in the Partition Manager, the Message Passing Interface (MPI), and (optionally) Low-level Applications Programming Interface (LAPI).

FLAGS

Any of the compiler flags normally accepted by the **xlc_r** or **cc_r** command can also be used on **mpcc_r**. For a complete listing of these flag options, refer to the manual page for the compiler **cc_r** command. Typical options to **mpcc_r** include:

-v causes a "verbose" output listing of the shell script.

-g produces an object file with symbol table references.

-o names the executable.

-l (lowercase L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the CONTEXT section.

Note: Not all AIX libraries are thread safe. Verify that your intended use is supported.

-I (uppercase i)
names directories for additional includes. The directory */usr/lpp/ppe.poe/include* or the appropriate subdirectory is included automatically. Command line or makefile hard coding of include paths for PE header files should normally be avoided. Such specifications will take precedence over the directory selected by the script and may result in generating incorrect code.

-p
enables profiling with the **prof** command. For more information, see the appendix on profiling programs in *IBM Parallel Environment: Operation and Use*

-pg
enables profiling with the **xprofiler** and **gprof** commands. For more information, see the xprofiler information in *AIX Performance Tools Guide and Reference* or *AIX Performance Tools Guide and Reference* and the information on profiling programs with **prof** and **gprof** in *IBM Parallel Environment: Operation and Use*.

-q64
enables compiling of 64-bit applications.

DESCRIPTION

The **mpcc_r** shell script invokes the **xlc_r** command. In addition, the Partition Manager and data communication interfaces are automatically linked in. The script creates an executable that dynamically binds with the communication subsystem libraries.

Flags are passed by **mpcc_r** to the **xlc_r** command, so any of the **xlc_r** options can be used on the **mpcc_r** shell script. The communication subsystem library implementation is dynamically linked when you invoke the executable using the **poe** command. The value specified by the **MP_EUILIB** environment variable or the **-euilib** flag will then determine which communication subsystem library implementation is dynamically linked.

ENVIRONMENT VARIABLES

MP_PREFIX
: Sets an alternate path to the scripts library. If not set or NULL, the standard path */usr/lpp/ppe.poe* is used. If this environment variable is set, then all libraries are prefixed by *$MP_PREFIX/ppe.poe*.

OBJECT_MODE
: Setting this variable to *64* causes the 64–bit libraries to be linked to the executable, as if the **-q64** option had been set. If set to anything other than *64* or if not set, it will not affect how the executables are built.

EXAMPLES

To compile a C program, enter:

```
mpcc_r program.c -o program
```

FILES

When you compile a program using **mpcc_r**, the following libraries are automatically selected:

- /usr/lpp/ppe.poe/lib/libmpi_r.a (Message Passing Interface, collective communication routines)
- /usr/lpp/ppe.poe/lib/libppe_r.a (PE common routines)
- The following library is selected if it exists as a symbolic link to /opt/rsct/lapi/lib/liblapi_r.a:

  ```
  /usr/lib/liblapi_r.a
  ```

RELATED INFORMATION

Commands: **mpCC_r**(1), **cc**(1)

mpCC

Invokes a C++ compiler to compile C++ programs that use MPI.

SYNOPSIS

mpCC [*c++_flags*]... *program.C*

The **mpCC** shell script compiles C++ programs while linking in the Partition Manager, the Message Passing Interface (MPI), and Low-level Applications Programming Interface (LAPI).

You can specify a C ++ compiler with the **MP_COMPILER** environment variable. If the specified compiler is one of the C++ compilers supported by the IBM C **Set++ vacpp.cmp** package or is the GNU g++ compiler, it is invoked using its default install path. It is assumed that any other compiler is reachable via the user's regular search path.

If **MP_COMPILER** is not defined, the **mpCC** shell script first searches for the xlC_r compiler of the IBM C **Set++ vac.cmp** package. If the IBM C **Set++ vac.cmp** package is not installed, it looks for the GNU g++ compiler.

FLAGS

The **mpCC** shell script passes most flags directly to the compiler. Some flags are interrupted by the **mpCC** shell script. These are:

-h prints a help message.

-v is passed to the compiler and causes a *verbose* output listing of the shell script.

-m32 | -q32
enables compiling of 32-bit applications (default). The **q** flag is used by the IBM compiler and the **m** flag is used by the GNU g++ compiler. The **mpCC** script accepts them interchangeably and passes the right flag to the right compiler.

-m64 | -q64
enables compiling of 64-bit applications. The **q** flag is used by the IBM compiler and the **m** flag is used by the GNU g++ compiler. The **mpCC** script accepts them interchangeably and passes the right flag to the right compiler.

-compiler
specifies the name of the C compiler to use. The default value is **xlc_r**, if the IBM C **Set++ vac.cmp** package is installed. If it is not installed, the default value becomes **gcc** (the GNU GCC compiler).

You can use the **-compiler** flag to specify a compiler other than **xlc_r**. For example, you may want to use the GNU GCC compiler, even if the IBM C **Set++ vac.cmp** package is installed. Or, you may want to specify one of the other compilers that are contained in the IBM C **Set++ vac.cmp** package.

To specify a third-party compiler, you must either specify its full path or add that compiler to a directory in your search path (for example, **/usr/bin**).

-g Produces an object file with symbol table references. This object file is needed for debugging with the GNU GDB debugger.

-o names the executable.

-cpp
: enables the use of full C++ bindings in MPI.

-l (lowercase L)
: names additional libraries to be searched. Several libraries are automatically included, and are listed below in the *Context* section of this manual page.

-I (uppercase i)
: names directories for additional includes. The directory **/opt/ibmhpc/ppe.poe/include** or the appropriate subdirectory is included automatically. Command line or makefile hard-coding of include paths for PE header files should normally be avoided. Such specifications will take precedence over the directory selected by the script and may cause incorrect code to be generated.

-pg
: enables profiling with the **gprof** command. For more information, see the information on profiling programs in *IBM Parallel Environment: Operation and Use*.

DESCRIPTION

The **mpCC** shell script invokes a C++ compiler to compile C++ programs that use MPI. In addition, the Partition Manager and data communication interfaces are automatically linked in. The script creates an executable that dynamically binds with the communication subsystem libraries.

You can specify a C++ compiler with the **MP_COMPILER** environment variable.

Most C++ compiler flags are passed directly from **mpCC** to the compiler. The communication subsystem library implementation is dynamically linked when you invoke the executable using the **poe** command.

ENVIRONMENT VARIABLES

MP_COMPILER
: Specify the name of the C++ compiler to use. The default value is **xlC_r**, if the IBM C **Set++ vac.cmp** package is installed. If it is not installed, the default value becomes **g++** (the GNU g++ compiler).

: You can use **MP_COMPILER** to specify a compiler other than **xlC_r**. For example, you may want to use the GNU g++ compiler, even if the IBM C **Set++ vac.cmp** package is installed. Or, you may want to specify one of the other compilers that are contained in the IBM C **Set++ vac.cmp** package.

: To specify a third-party compiler, you must either specify its full path or add that compiler to a directory in your search path (for example, **/usr/bin**).

MP_PREFIX
: Sets an alternate path to the default IBM PE library and include file path. If not set or NULL, the default library path is **/usr/lib**, and the default include file path is **/opt/ibmhpc/ppe.poe/include**. If this environment variable is set, then all library search paths provided by **mpCC** are prefixed by **$MP_PREFIX/lib**, and all include file search paths are prefixed by **$MP_PREFIX/include**.

OBJECT_MODE
: Setting this variable to 64 causes the 64-bit libraries to be linked to the executable, as if the **-q64** option had been set. If you do not set

OBJECT_MODE, or if you set it to anything other than *64* , the 32-bit libraries will be linked to the executable.

EXAMPLES

To compile a C++ program, enter:

```
mpCC program.C -o program
```

FILES

When you compile a program using **mpCC**, the following libraries are automatically selected.

- Message passing interface, collective communication routines:
  - **/usr/lib/libmpi_ibm.so**
  - **/usr/lib/libpoe.so**
  - **/usr/lib64/libmpi_ibm.so**
  - **/usr/lib64/libpoe.so**
- IBM Low-Level Applications Programming Interface routines:
  - **/usr/lib/liblapi.so**
  - **/usr/lib64/liblapi.so**

RELATED INFORMATION

Commands: **mpcc**(1), **mpfort**(1)

mpCC_r

Invokes a shell script to compile C++ programs that use MPI. **This command applies to PE for AIX only.**

SYNOPSIS

mpCC_r [*xlC_flags*]... *program.C*

The **mpCC_r** shell script compiles C++ programs while linking in the Partition Manager, Message Passing Interface (MPI), and (optionally) Low-level Applications Programming Interface (LAPI).

FLAGS

Any of the compiler flags normally accepted by the **xlC_r** command can also be used on **mpCC_r**. For a complete listing of these flag options, refer to the manual page for the **xlC_r** command. Typical options to **mpCC_r** include:

-v causes a "verbose" output listing of the shell script.

-g produces an object file with symbol table references.

-o names the executable.

-cpp
enables the use of full C++ bindings in MPI.

-l (lowercase L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the CONTEXT section.

Note: Not all AIX libraries are thread safe. Verify that your intended use is supported.

-I (uppercase i)
names directories for additional includes. The directory */usr/lpp/ppe.poe/include* or the appropriate subdirectory is included automatically. Command line or makefile hard coding of include paths for PE header files should normally be avoided. Such specifications will take precedence over the directory selected by the script and may result in generating incorrect code.

-p
enables profiling with the **prof** command. For more information, see *IBM Parallel Environment: Operation and Use*.

-pg
enables profiling with the **xprofiler** and **gprof** commands. For more information, see the xprofiler information in *AIX Performance Tools Guide and Reference* and the information about profiling programs with **prof** and **gprof** in *IBM Parallel Environment: Operation and Use*.

-q64
enables compiling of 64-bit applications.

DESCRIPTION

The **mpCC_r** shell script invokes the **xlC_r** command. In addition, the Partition Manager and data communication interfaces are automatically linked in. The script creates an executable that dynamically binds with the communication subsystem libraries.

Flags are passed by **mpCC_r** to the **xlC_r** command, so any of the **xlC_r** options can be used on the **mpCC_r** shell script. The communication subsystem library implementation is dynamically linked when you invoke the executable using the **poe** command. The value specified by the **MP_EUILIB** environment variable or the **-euilib** flag will then determine which communication subsystem library implementation is dynamically linked.

ENVIRONMENT VARIABLES

MP_PREFIX
: Sets an alternate path to the scripts library. If not set or NULL, the standard path */usr/lpp/ppe.poe* is used. If this environment variable is set, then all libraries are prefixed by *$MP_PREFIX/ppe.poe*.

OBJECT_MODE
: Setting this variable to *64* causes the 64–bit libraries to be linked to the executable, as if the **-q64** option had been set. If set to anything other than *64* or if not set, it will not affect how the executables are built.

EXAMPLES

To compile a C++ program, enter:

```
mpCC_r program.C -o program
```

FILES

When you compile a program using **mpCC_r**, the following libraries are automatically selected:

- /usr/lpp/ppe.poe/lib/libmpi_r.a (Message passing interface, collective communication routines)
- /usr/lpp/ppe.poe/lib/libppe_r.a (PE common routines)
- The following library is selected if it exists as a symbolic link to /opt/rsct/lapi/lib/liblapi_r.a:

  ```
  /usr/lib/liblapi_r.a
  ```

RELATED INFORMATION

Commands: **mpcc_r**(1), **xlC**(1)

mpfort

Invokes a Fortran compiler to compile Fortran programs that use MPI. This command applies to PE for Linux only.

SYNOPSIS

mpfort [*cc_flags*]... *program.F*

The **mpfort** shell script compiles Fortran programs while linking in the Partition Manager, the Message Passing Interface (MPI), and Low-level Applications Programming Interface (LAPI).

You can specify a Fortran compiler with the **MP_COMPILER** environment variable. If the specified compiler is one of the Fortran compilers supported by the IBM C **Set++ xlf.cmp** package or is the GNU g77 compiler, it is invoked using its default install path. It is assumed that any other compiler is reachable via the user's regular search path.

If **MP_COMPILER** is not defined, the **mpfort** shell script first searches for the xlf_r compiler of the IBM C **Set++ xlf.cmp** package. If the IBM C **Set++ xlf.cmp** package is not installed, it looks for the GNU g77 compiler.

Beginning with Version 5.1, PE supports Fortran 90 *modules*. PE now includes a Fortran 90 module (**mpi.mod**) that provides type checking for MPI programs at compile time. This allows programmers to find and resolve errors at a much earlier stage.

To use the PE Fortran 90 type-checking module, do the following.

For new programs:
- Include the `USE MPI` statement to the application source code.
- Compile the program with XLF Version 12.1 or later.

For existing programs:
- Modify the application source code to include the `USE MPI` statement.
- Recompile the program with XL Fortran compiler (XLF) Version 12.1 or later. Note that if you have an existing Fortran MPI program that is running without errors, there is no need to recompile it. Since the program is already running correctly, compiling it with the Fortran 90 module would provide no benefit.

Note: The **mpi.mod** module was compiled without the **-qmixed** compiler option. As a result, routines that use the **mpi.mod** module must not be compiled with this option.

Note: The **USE MPI** statement is supported only in conjunction with IBM's XLF compiler.

FLAGS

The **mpfort** shell script passes most flags directly to the compiler. Some flags are interrupted by the **mpfort** shell script. These are:

-h prints a help message.

-v is passed to the compiler and causes a *verbose* output listing of the shell script.

-m32 | -q32
enables compiling of 32-bit applications (default). The **q** flag is used by the IBM compiler and the **m** flag is used by the GNU g77 compiler. The **mpfort** script accepts them interchangeably and passes the right flag to the right compiler.

-m64 | -q64
enables compiling of 64-bit applications. The **q** flag is used by the IBM compiler and the **m** flag is used by the GNU g77 compiler. The **mpfort** script accepts them interchangeably and passes the right flag to the right compiler.

-compiler
specifies the name of the C compiler to use. The default value is **xlf_r**, if the IBM C **Set++ xlf.cmp** package is installed. If it is not installed, the default value becomes **gcc** (the GNU GCC compiler).

You can use the **-compiler** flag to specify a compiler other than **xlf_r**. For example, you may want to use the GNU GCC compiler, even if the IBM C **Set++ xlf.cmp** package is installed. Or, you may want to specify one of the other compilers that are contained in the IBM C **Set++ xlf.cmp** package.

To specify a third-party compiler, you must either specify its full path or add that compiler to a directory in your search path (for example, **/usr/bin**).

Other commonly used Fortran compiler flags are:

-g Produces an object file with symbol table references. This object file is needed for debugging with the GNU GDB debugger.

-o names the executable.

-l (lowercase L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the *Context* section of this manual page.

-I (uppercase i)
names directories for additional includes. The directory **/opt/ibmhpc/ppe.poe/include** or the appropriate subdirectory is included automatically. Command line or makefile hard-coding of include paths for PE header files should normally be avoided. Such specifications will take precedence over the directory selected by the script and may cause incorrect code to be generated.

-pg
enables profiling with **gprof** command. For more information, see the information on profiling programs in *IBM Parallel Environment: Operation and Use*.

DESCRIPTION

The **mpfort** shell script invokes a Fortran compiler to compile Fortran programs that use MPI. In addition, the Partition Manager and data communication interfaces are automatically linked in. The script creates an executable that dynamically binds with the communication subsystem libraries.

You can specify a Fortran compiler with the **MP_COMPILER** environment variable.

Most Fortran compiler flags are passed directly from **mpfort** to the compiler. The communication subsystem library implementation is dynamically linked when you invoke the executable using the **poe** command.

ENVIRONMENT VARIABLES

MP_COMPILER
Specifies the name of the Fortran compiler to use. The default value is **xlf_r**, if the IBM C **Set++ xlf.cmp** package is installed. If it is not installed, the default value becomes **g77** (the GNU G77 compiler).

You can use **MP_COMPILER** to specify a compiler other than **xlf_r**. For example, you may want to use the GNU G77 compiler, even if the IBM C **Set++ xlf.cmp** package is installed. Or, you may want to specify one of the other compilers that are contained in the IBM C **Set++ xlf.cmp** package.

To specify a third-party compiler, you must either specify its full path or add that compiler to a directory in your search path (for example, **/usr/bin**).

MP_PREFIX
Sets an alternate path to the default IBM PE library and include file path. It not set or NULL, the default library path is **/usr/lib**, and the default include file path is**/opt/ibmhpc/ppe.poe/include**. If this environment variable is set, then all library search paths provided by **mpfort** are prefixed by **$MP_PREFIX/lib**, and all include file search paths are prefixed by **$MP_PREFIX/include**.

OBJECT_MODE
Setting this variable to 64 causes the 64-bit libraries to be linked to the executable, as if the **-q64** option had been set. If you do not set **OBJECT_MODE**, or if you set it to anything other than *64* , the 32-bit libraries will be linked to the executable.

EXAMPLES

To compile a Fortran program, enter:

```
mpfort program.c -o program
```

FILES

When you compile a program using **mpfort**, the following libraries are automatically selected.

- Message passing interface, collective communication routines:
  - **/usr/lib/libmpi_ibm.so**
  - **/usr/lib/libpoe.so**
  - **/usr/lib64/libmpi_ibm.so**
  - **/usr/lib64/libpoe.so**
- IBM Low-Level Applications Programming Interface routines:
  - **/usr/lib/liblapi.so**
  - **/usr/lib64/liblapi.so**

RELATED INFORMATION

Commands: **mpcc**(1), **mpCC**(1)

mpiexec

Invokes the Parallel Operating Environment (POE) for loading and executing programs on remote processor nodes. This command invokes the **poe** command.

SYNOPSIS

mpiexec -n *partition_size program*

The **mpiexec** command is described in the MPI-2 standard as a portable way of starting MPI jobs; it is provided here to conform with that standard. The **mpiexec** command invokes **poe** to run the specified *program*. The **mpiexec** command translates the **-n** flag to the **-procs** flag for the **poe** command. The **mpiexec** command passes all other arguments unchanged to the **poe** command. Refer to the **poe** command man page for additional details on its flags.

Note: Under PE for Linux, if another MPI package is installed on the node before you install PE, **mpiexec** will not be linked to the **/usr/bin** directory. In this case, the full path for accessing **mpiexec** is**/opt/ibmhpc/ppe.poe/bin/mpiexec**.

FLAGS

In addition to the **-n** flag described below, all **poe** command flags are accepted, and passed unchanged to the **poe** command.

If you are familiar with the description of the **mpiexec** command in the MPI-2 standard, please note that we have chosen to implement only the command syntax required for compliance with that standard. The optional flags have not been implemented, as our **poe** command, which is invoked by the **mpiexec** command, offers sufficient functionality.

-n
Translated to the **-procs** flag and passed to the **poe** command. This determines the number of program tasks. If not set, the default is 1.

EXAMPLES

To invoke an MPI program *sample* to run as five tasks:

```
mpiexec -n 5 sample
```

RELATED INFORMATION

Commands: **poe**(1)

mpxlf_r

Invokes a shell script to compile Fortran programs which use MPI. **This command applies to PE for AIX only.**

SYNOPSIS

mpxlf_r [*xlf_flags*]... *program.f*

The **mpxlf_r** shell script compiles Fortran programs while linking in the Partition Manager, the Message Passing Interface (MPI), and (optionally) Low-level Applications Programming Interface (LAPI).

FLAGS

Any of the compiler flags normally accepted by the **xlf** command can also be used on **mpxlf_r**. For a complete listing of these flag options, refer to the manual page for the **xlf** command. Typical options to **mpxlf_r** include:

-v causes a "verbose" output listing of the shell script.

-g produces an object file with symbol table references.

-o names the executable.

-l (lowercase L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the CONTEXT section.

Note: Not all AIX libraries are thread safe. Verify that your intended use is supported.

-I (uppercase i)
names directories for additional includes. The directory */usr/lpp/ppe.poe/include* or the appropriate subdirectory is included automatically. Command line or makefile hard coding of include paths for PE header files should normally be avoided. Such specifications will take precedence over the directory selected by the script and may result in generating incorrect code.

-p
enables profiling with the **prof** command. For more information, see *IBM Parallel Environment: Operation and Use*.

-pg
enables profiling with the **xprofiler** and **gprof** commands. For more information, see the xprofiler information in *AIX Performance Tools Guide and Reference* and the information on profiling programs with **prof** and **gprof** in *IBM Parallel Environment: Operation and Use*.

-q64
Causes code to compile to 64-bit objects. On AIX, the default is to compile to 32-bit objects. The **OBJECT_MODE** environment variable can be used to select either a 32-bit or 64-bit default. **–q64** supersedes any **OBJECT_MODE** setting.

-q32
Causes code to compile to 32-bit objects. On AIX, the default is to compile to 32-bit objects. **–q32** is required only if the **OBJECT_MODE** environment variable has been used to change the default. **–q32** supersedes any **OBJECT_MODE** setting.

DESCRIPTION

The **mpxlf_r** shell script invokes the **xlf** command. In addition, the Partition Manager and data communication interfaces are automatically linked in. The script creates an executable that dynamically binds with the communication subsystem libraries.

Flags are passed by **mpxlf_r** to the **xlf** command, so any of the **xlf** options can be used on the **mpxlf_r** shell script. The communication subsystem library implementation is dynamically linked when you invoke the executable using the **poe** command. The value specified by the **MP_EUILIB** environment variable or the **-euilib** flag will then determine which communication subsystem library implementation is dynamically linked.

There are distinct 32-bit and 64-bit versions of **mpif.h** and **mpi.mod**, and the Fortran compilation scripts provide the include path to select the correct version. The script's decision is based on the **OBJECT_MODE** environment variable setting and the use of the **–q32** or **-q64** flags when the script was invoked. Alternate ways of forcing 32-bit or 64-bit compilation may result in selecting the wrong include. A user-specified include path provided through a makefile or compilation command line flag will be searched before the script's path. If any user-specified include path provides an inappropriate copy of *mpif.h*, the script will not be able to override and select the appropriate copy. Alterations made to **xlf.cfg**, in an effort to force 64-bit compilation, are not recognized by the script.

ENVIRONMENT VARIABLES

MP_PREFIX
: Sets an alternate path to the scripts library. If not set or NULL, the standard path */usr/lpp/ppe.poe* is used. If this environment variable is set, then all libraries are prefixed by *$MP_PREFIX/ppe.poe*.

OBJECT_MODE
: Setting this variable to *64* causes the 64–bit libraries to be linked to the executable, as if the **-q64** option had been set. Setting this variable to anything other than 64, or not setting it has no effect on how the executables are built.

EXAMPLES

To compile a Fortran program, enter:

```
mpxlf_r program.f -o program
```

FILES

When you compile a program using **mpxlf_r**, the following libraries are automatically selected:

- /usr/lpp/ppe.poe/lib/libmpi_r.a (Message passing interface, collective communication routines)
- /usr/lpp/ppe.poe/lib/libppe_r.a (PE common routines)
- The following library is selected if it exists as a symbolic link to /opt/rsct/lapi/lib/liblapi_r.a:

  ```
  /usr/lib/liblapi_r.a
  ```

RELATED INFORMATION

Commands: **mpcc_r**(1), **xlf_r**(1)

mpxlf90_r

Invokes a shell script to compile Fortran 90 programs which use MPI. **This command applies to PE for AIX only.**

SYNOPSIS

mpxlf90_r [*xlf_flags*]... *program.f*

The **mpxlf90_r** shell script compiles Fortran 90 programs while linking in the Partition Manager, the Message Passing Interface (MPI), and (optionally) Low-level Applications Programming Interface (LAPI).

Beginning with Version 5.1, PE supports Fortran 90 *modules*. PE now includes a Fortran 90 module (**mpi.mod**) that provides type checking for MPI programs at compile time. This allows programmers to find and resolve errors at a much earlier stage.

To use the PE Fortran 90 type-checking module, do the following.

For new programs:
- Include the `USE MPI` statement to the application source code.
- Compile the program with XLF Version 12.1 or later.

For existing programs:
- Modify the application source code to include the `USE MPI` statement.
- Recompile the program with XL Fortran compiler (XLF) Version 12.1 or later. Note that if you have an existing Fortran MPI program that is running without errors, there is no need to recompile it. Since the program is already running correctly, compiling it with the Fortran 90 module would provide no benefit.

Note: The **mpi.mod** module was compiled without the **-qmixed** compiler option. As a result, routines that use the **mpi.mod** module must not be compiled with this option.

Note: The **USE MPI** statement is supported only in conjunction with IBM's XLF compiler.

FLAGS

Any of the compiler flags normally accepted by the **xlf** command can also be used on **mpxlf90_r**. For a complete listing of these flag options, refer to the manual page for the **xlf** command. Typical options to **mpxlf90_r** include:

-v causes a "verbose" output listing of the shell script.

-g produces an object file with symbol table references.

-o names the executable.

-l (lowercase L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the CONTEXT section.

Note: Not all AIX libraries are thread safe. Verify that your intended use is supported.

-I (uppercase i)
names directories for additional includes. The directory */usr/lpp/ppe.poe/include* or the appropriate subdirectory is included automatically. Command line or makefile hard coding of include paths for PE header files should normally be avoided. Such specifications will take precedence over the directory selected by the script and may result in generating incorrect code.

-p
enables profiling with the **prof** command. For more information, see *IBM Parallel Environment: Operation and Use*.

-pg
enables profiling with the **xprofiler** and **gprof** commands. For more information, see the xprofiler information in *AIX Performance Tools Guide and Reference* and the information on profiling programs with **prof** and **gprof** in *IBM Parallel Environment: Operation and Use*.

-q64
Causes code to compile to 64-bit objects. On AIX, the default is to compile to 32-bit objects. The **OBJECT_MODE** environment variable can be used to select either a 32-bit or 64-bit default. **–q64** supersedes any **OBJECT_MODE** setting.

-q32
Causes code to compile to 32-bit objects. On AIX, the default is to compile to 32-bit objects. **–q32** is required only if the **OBJECT_MODE** environment variable has been used to change the default. **–q32** supersedes any **OBJECT_MODE** setting.

DESCRIPTION

The **mpxlf90_r** shell script invokes the **xlf** command. In addition, the Partition Manager and data communication interfaces are automatically linked in. The script creates an executable that dynamically binds with the communication subsystem libraries.

Flags are passed by **mpxlf90_r** to the **xlf** command, so any of the **xlf** options can be used on the **mpxlf90_r** shell script. The communication subsystem library implementation is dynamically linked when you invoke the executable using the **poe** command. The value specified by the **MP_EUILIB** environment variable or the **-euilib** flag will then determine which communication subsystem library implementation is dynamically linked.

There are distinct 32-bit and 64-bit versions of **mpif.h** and **mpi.mod**, and the Fortran compilation scripts provide the include path to select the correct version. The script's decision is based on the **OBJECT_MODE** environment variable setting and the use of the **–q32** or **-q64** flags when the script was invoked. Alternate ways of forcing 32-bit or 64-bit compilation may result in selecting the wrong include. A user specified include path provided through a makefile or compilation command line flag will be searched before the script's path. If any user-specified include path provides an inappropriate copy of *mpif.h*, the script will not be able to override and select the appropriate copy. Alterations made to **xlf.cfg**, in an effort to force 64-bit compilation, are not recognized by the script.

ENVIRONMENT VARIABLES

MP_PREFIX
Sets an alternate path to the scripts library. If not set or NULL, the

standard path */usr/lpp/ppe.poe* is used. If this environment variable is set, then all libraries are prefixed by *$MP_PREFIX/ppe.poe*.

OBJECT_MODE
Setting this variable to *64* causes the 64–bit libraries to be linked to the executable, as if the **-q64** option had been set. Setting this variable to anything other than 64, or not setting it has no effect on how the executables are built.

EXAMPLES

To compile a Fortran 90 program, enter:

```
mpxlf90_r program.f -o program
```

FILES

When you compile a program using **mpxlf90_r**, the following libraries are automatically selected:

- /usr/lpp/ppe.poe/lib/libmpi_r.a (Message passing interface, collective communication routines)
- /usr/lpp/ppe.poe/lib/libppe_r.a (PE common routines)
- The following library is selected if it exists as a symbolic link to /opt/rsct/lapi/lib/liblapi_r.a:

  ```
  /usr/lib/liblapi_r.a
  ```

RELATED INFORMATION

Commands: **mpcc_r**(1), **xlf_r**(1), **mpxlf_r**(1)

mpxlf95_r

Invokes a shell script to compile Fortran 95 programs which use MPI. **This command applies to PE for AIX only.**

SYNOPSIS

mpxlf95_r [*xlf_flags*]... *program.f*

The **mpxlf95_r** shell script compiles Fortran 95 programs while linking in the Partition Manager, the Message Passing Interface (MPI), and (optionally) Low-level Applications Programming Interface (LAPI).

FLAGS

Any of the compiler flags normally accepted by the **xlf95** command can also be used on **mpxlf95_r**. For a complete listing of these flag options, refer to the manual page for the **xlf95** command. Typical options to **mpxlf95_r** include:

-v causes a "verbose" output listing of the shell script.

-g produces an object file with symbol table references.

-o names the executable.

-l (lowercase L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the CONTEXT section.

Note: Not all AIX libraries are thread safe. Verify that your intended use is supported.

-I (uppercase i)
names directories for additional includes. The directory */usr/lpp/ppe.poe/include* or the appropriate subdirectory is included automatically. Command line or makefile hard coding of include paths for PE header files should normally be avoided. Such specifications will take precedence over the directory selected by the script and may result in generating incorrect code.

-p
enables profiling with the **prof** command. For more information, see *IBM Parallel Environment: Operation and Use*.

-pg
enables profiling with the **xprofiler** and **gprof** commands. For more information, see the xprofiler information in *AIX Performance Tools Guide and Reference* and the information on profiling programs with **prof** and **gprof** in *IBM Parallel Environment: Operation and Use*.

-q64
Causes code to compile to 64-bit objects. On AIX, the default is to compile to 32-bit objects. The **OBJECT_MODE** environment variable can be used to select either a 32-bit or 64-bit default. **–q64** supersedes any **OBJECT_MODE** setting.

-q32
Causes code to compile to 32-bit objects. On AIX, the default is to compile to 32-bit objects. **–q32** is required only if the **OBJECT_MODE** environment variable has been used to change the default. **–q32** supersedes any **OBJECT_MODE** setting.

DESCRIPTION

The **mpxlf95_r** shell script invokes the **xlf95** command. In addition, the Partition Manager and data communication interfaces are automatically linked in. The script creates an executable that dynamically binds with the communication subsystem libraries.

Flags are passed by **mpxlf95_r** to the **xlf95** command, so any of the **xlf95** options can be used on the **mpxlf95_r** shell script. The communication subsystem library implementation is dynamically linked when you invoke the executable using the **poe** command. The value specified by the **MP_EUILIB** environment variable or the **-euilib** flag will then determine which communication subsystem library implementation is dynamically linked.

There are distinct 32-bit and 64-bit versions of **mpif.h** and **mpi.mod**, and the Fortran compilation scripts provide the include path to select the correct version. The script's decision is based on the **OBJECT_MODE** environment variable setting and the use of the **–q32** or **-q64** flags when the script was invoked. Alternate ways of forcing 32-bit or 64-bit compilation may result in selecting the wrong include. A user-specified include path provided through a makefile or compilation command line flag will be searched before the script's path. If any user-specified include path provides an inappropriate copy of *mpif.h*, the script will not be able to override and select the appropriate copy. Alterations made to **xlf.cfg**, in an effort to force 64-bit compilation, are not recognized by the script.

ENVIRONMENT VARIABLES

MP_PREFIX
: Sets an alternate path to the scripts library. If not set or NULL, the standard path */usr/lpp/ppe.poe* is used. If this environment variable is set, then all libraries are prefixed by *$MP_PREFIX/ppe.poe*.

OBJECT_MODE
: Setting this variable to *64* causes the 64–bit libraries to be linked to the executable, as if the **-q64** option had been set. Setting this variable to anything other than 64, or not setting it has no effect on how the executables are built.

EXAMPLES

To compile a Fortran 95 program, enter:

```
mpxlf95_r program.f -o program
```

FILES

When you compile a program using **mpxlf95_r**, the following libraries are automatically selected:

- /usr/lpp/ppe.poe/lib/libmpi_r.a (Message passing interface, collective communication routines)
- /usr/lpp/ppe.poe/lib/libppe_r.a (PE common routines)
- The following library is selected if it exists as a symbolic link to /opt/rsct/lapi/lib/liblapi_r.a:

  ```
  /usr/lib/liblapi_r.a
  ```

RELATED INFORMATION

Commands: **mpcc_r**(1), **xlf95_r**(1), **mpxlf_r**(1), **mpxlf95**(1)

mpxlf2003_r

Invokes a shell script to compile Fortran 2003 programs which use MPI. **This command applies to PE for AIX only.**

SYNOPSIS

mpxlf2003_r [*xlf_flags*]... *program.f*

The **mpxlf2003_r** shell script compiles Fortran 2003 programs while linking in the Partition Manager, the Message Passing Interface (MPI), and (optionally) Low-level Applications Programming Interface (LAPI).

FLAGS

Any of the compiler flags normally accepted by the **xlf2003_r** command can also be used on **mpxlf2003_r**. For a complete listing of these flag options, refer to the manual page for the **xlf2003_r** command. Typical options to **mpxlf2003_r** include:

-v causes a "verbose" output listing of the shell script.

-g produces an object file with symbol table references.

-o names the executable.

-l (lowercase L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the CONTEXT section.

Note: Not all AIX libraries are thread safe. Verify that your intended use is supported.

-I (uppercase i)
names directories for additional includes. The directory */usr/lpp/ppe.poe/include* or the appropriate subdirectory is included automatically. Command line or makefile hard coding of include paths for PE header files should normally be avoided. Such specifications will take precedence over the directory selected by the script and may result in generating incorrect code.

-p
enables profiling with the **prof** command. For more information, see the information on profiling programs in *IBM Parallel Environment: Operation and Use*.

-pg
enables profiling with the **xprofiler** and **gprof** commands. For more information, see the xprofiler information in *AIX Performance Tools Guide and Reference* or the information on profiling programs with **prof** and **gprof** in *IBM Parallel Environment: Operation and Use*.

-q64
Causes code to compile to 64-bit objects. On AIX, the default is to compile to 32-bit objects. The **OBJECT_MODE** environment variable can be used to select either a 32-bit or 64-bit default. **–q64** supersedes any **OBJECT_MODE** setting.

-q32
Causes code to compile to 32-bit objects. On AIX, the default is to compile to

32-bit objects. **–q32** is required only if the **OBJECT_MODE** environment variable has been used to change the default. **–q32** supersedes any **OBJECT_MODE** setting.

DESCRIPTION

The **mpxlf2003_r** shell script invokes the **xlf2003_r** command. In addition, the Partition Manager and data communication interfaces are automatically linked in. The script creates an executable that dynamically binds with the communication subsystem libraries.

Flags are passed by **mpxlf2003_r** to the **xlf2003_r** command, so any of the **xlf2003_r** options can be used on the **mpxlf2003_r** shell script. The communication subsystem library implementation is dynamically linked when you invoke the executable using the **poe** command. The value specified by the **MP_EUILIB** environment variable or the **-euilib** flag will then determine which communication subsystem library implementation is dynamically linked.

There are distinct 32-bit and 64-bit versions of **mpif.h** and **mpi.mod**, and the Fortran compilation scripts provide the include path to select the correct version. The script's decision is based on the **OBJECT_MODE** environment variable setting and the use of the **–q32** or **-q64** flags when the script was invoked. Alternate ways of forcing 32-bit or 64-bit compilation may result in selecting the wrong include. A user-specified include path provided through a makefile or compilation command line flag will be searched before the script's path. If any user-specified include path provides an inappropriate copy of *mpif.h*, the script will not be able to override and select the appropriate copy. Alterations made to **xlf.cfg**, in an effort to force 64-bit compilation, are not recognized by the script.

ENVIRONMENT VARIABLES

MP_PREFIX
Sets an alternate path to the scripts library. If not set or NULL, the standard path */usr/lpp/ppe.poe* is used. If this environment variable is set, then all libraries are prefixed by *$MP_PREFIX/ppe.poe*.

OBJECT_MODE
Setting this variable to *64* causes the 64–bit libraries to be linked to the executable, as if the **-q64** option had been set. Setting this variable to anything other than 64, or not setting it has no effect on how the executables are built.

EXAMPLES

To compile a Fortran 2003 program, enter:

```
mpxlf2003_r program.f -o program
```

FILES

When you compile a program using **mpxlf2003_r**, the following libraries are automatically selected:

- /usr/lpp/ppe.poe/lib/libmpi_r.a (Message passing interface, collective communication routines)
- /usr/lpp/ppe.poe/lib/libppe_r.a (PE common routines)

- The following library is selected if it exists as a symbolic link to /opt/rsct/lapi/lib/liblapi_r.a:

```
/usr/lib/liblapi_r.a
```

RELATED INFORMATION

Commands: **mpcc_r**(1), **xlf2003_r**(1), **mpxlf_r**(1), **mpxlf90_r**(1), **mpxlf95_r**

pdb

Invokes the PDB parallel debugger.

SYNOPSIS

To invoke PDB in launch mode:

pdb *executable_and_arguments*
[**--poe** *poe_options*]
[**--gdb** *gdb_options*] (Linux only) or [**--dbx** *dbx_options*] (AIX only)

To invoke PDB in attach mode:

pdb **-a** [*poe_process_id*]
[**--gdb** *gdb_options*] (Linux only) or [**--dbx** *dbx_options*] (AIX only)

To get PDB help:

pdb [**-h**]

The **pdb** command launches PDB, the Parallel Environment's command line debugger for parallel programs.

FLAGS

executable_and_arguments
Specifies to launch the target executable and start debugging from the beginning of the program. The arguments are to the executable, not to PDB.

--dbx *dbx_options*
Specifies the options to pass to dbx. Applies to AIX only.

-a [*poe_process_id*]
Specifies to attach to a running POE job. **pdb -a** must be issued from the node on which the POE job was initiated. If the POE process ID is not specified, PDB tries to find one and reports an error if multiple POE processes exist.

-h Provides help information.

--gdb *gdb_options*
Specifies the options to pass to GDB. Applies to Linux only.

--poe *poe_options*
Specifies the options to pass to POE.

DESCRIPTION

PDB is the Parallel Environment's command line debugger for parallel programs. It works together with DISH, a tool for launching and managing distributed processes interactively, as well as GDB, the GNU project debugger and dbx, a UNIX-based debugger.. All of the commands you use with PDB are either DISH commands, for process management, or GDB or dbx commands, for serial debugging. PDB simply configures DISH to manage distributed GDB or dbx instances and then hands control over to DISH.

Debugging with PDB is similar to using multiple GDB or dbx instances to debug multiple processes simultaneously. PDB adds convenience by providing a control center for managing processes (DISH).

Before you can use PDB, you first need to compile the program you want to debug. You also need to be able to use POE to launch a parallel job with the compiled program.

PDB allows you to debug programs in two different modes; *launch mode* and *attach mode*. In launch mode, PDB launches the job and starts interactive debugging from the beginning of the program. In attach mode, PDB attaches to a program that is already in progress.

To use PDB in launch mode, precede the executable in your launch script with **pdb** on the command line. For example:

```
pdb my_program
```

To attach to a running job, enter **pdb -a** from the node on which the POE process is running.

When diagnosing problems, look at GDB or dbx first. For example, if you see some unexpected behavior while running PDB, you could issue one of the following commands to see if the same behavior exists under **rsh**:

```
rsh host gdb options program
```

Or

```
rsh host dbx options program
```

To obtain more information from PDB, use the **dish -i** *info_level* option.

The following are known issues with PDB:

edit command hangs

- Reason - GDB and dbx launch an external editor to edit source files, which sometimes causes PDB to hang.
- Solution - Do not use the **edit** command. In the event that a user enters the **edit** command, do one of the following:
  - If you are using GDB, press **<Ctrl-c>** to enter the interrupt state, and then type the command **interrupt**.
  - If you are using dbx, press **<Ctrl-c>** to enter the interrupt state, and then enter the command **send :q** twice.

Spurious SIG32 stops the job

- Reason - **pthread_cancel**() calls a program that generates SIG32 when the program is being debugged by GDB.
- Solution - Use the GDB command **handle** *SIG32 nostop*. This command can be put into **.gdbinit** to avoid typing it in for each session.

POE shows: "ERROR: 0031-652 Error reading STDIN"

- Reason - PDB uses POE to launch DISH agents, and POE is started in the background without STDIN.
- Solution - Ignore this error.

PDB cannot pass an option to GDB, dbx, or POE if there is a space in the option

- Reason - Spaces are treated as word delimitors by the shell script with which PDB is written.
- Solution - Do not use spaces in options.

ENVIRONMENT VARIABLES

In launch mode, PDB passes all POE environment variables to POE and to individual tasks. Attach mode does not require POE environment variables to be set.

EXAMPLES

1. To start PDB, set up the execution environment as you would for the POE command, then enter the **pdb** command, followed by the program being debugged:
   ```
   pdb my_program
   ```
2. To pass DISH options, add them to the end of the command:
   ```
   pdb my_program --dish -i 3
   ```
3. To pass GDB options, add them to the end of the command:
   ```
   pdb my_program --dish -i 3 --gdb -d source_dir
   ```
4. To pass POE options, add them to the end of the command:
   ```
   pdb my_program --dish -i 3 --gdb -d source_dir --poe -procs 2
   ```

FILES

The following temporary files are created during PDB execution.

/tmp/.pdb_config.*pdb_process_id*
: DISH configuration file that tells DISH to launch GDB.

/tmp/.pdb_instances.*pdb_process_id*
: DISH instance file that contains the hosts on which to launch GDB. For attach mode, this file also contains the process IDs of the program to debug.

RELATED INFORMATION

Commands: **dish**(1), **disha**(1), **poe**(1), **gdb**(1)

perpms

Shows the RPMs that are installed with PE. **This command applies to PE for Linux only.**

SYNOPSIS

perpms [**-h**] [**-a**]

FLAGS

-h
Causes the **perpms** man page to be printed to stdout.

-a Displays all installed RPMs of the IBM High Performance Computing Suite. This includes all RPMs of the IBM Parallel Environment for Linux (PE), LoadLeveler (LL), and Reliable Scalable Cluster Technology (RSCT).

DESCRIPTION

The **perpms** command returns a list of the RPMs that are currently installed with PE. If you invoke **perpms** without any flags, only the PE and LAPI RPM information is displayed. You can use the **-a** flag to also include the RPMs that are installed with any of the other IBM High Performance Computing Suite products, such as LoadLeveler and Reliable Scalable Cluster Technology (RSCT), that are installed on your system.

EXAMPLES

1. To display the RPM levels of all IBM High Performance Computing Suite products:

   ```
   perpms -a
   ```

   The output looks similar to this:

   ```
   <<< Install IBM PE RPM(s) >>>
   ppe_ppc_base_32bit_sles900-4.2.1.0-0607a
   ppe_ppc_64bit_sles900-4.2.1.0-0607a
   <<< Install IBM LAPI RPM(s) >>>
   lapi_ppc_base_32bit_sles900-2.4.1.0-0607a
   lapi_ppc_64bit_sles900-2.4.1.0-0607a
   <<< Install IBM LoadL RPM(s) >>>
   LoadL-full-lib-SLES9-PPC-3.3.2.0-0
   LoadL-full-license-SLES9-PPC64-3.3.2.0-0
   LoadL-full-SLES9-PPC64-3.3.2.0-0
   <<< Install IBM RSCT RPMs >>>
   rsct.core.cimrm-2.4.5.0-06045
   rsct.basic-2.4.5.0-06045
   rsct.64bit-2.4.5.0-0
   rsct.core.utils-2.4.5.0-06045
   rsct.core-2.4.5.0-06045
   ```

2. To display only the PE RPM levels:

   ```
   perpms
   ```

   The output looks similar to this:

   ```
   ppe_ppc_base_32bit_sles900-4.2.1.0-0607a
   ppe_ppc_64bit_sles900-4.2.1.0-0607a
   lapi_ppc_base_32bit_sles900-2.4.1.0-0607a
   lapi_ppc_64bit_sles900-2.4.1.0-0607a
   ```

poe

Invokes the Parallel Operating Environment (POE) for loading and executing programs on remote processor nodes.

SYNOPSIS

poe [**-h**] [**-v**] [*program*] [*program_options*]...
[**-adapter_use** *adapter_specifier*]
[**-buffer_mem** {*preallocated_buffer_size* | *,maximum_buffer_size* | *preallocated_buffer_size,maximum_buffer_size*}]
[**-bulk_min_msg_size** *message_size*]
[**-cc_buf_mem** {*preallocated_buffer_size* | *,maximum_buffer_size* | *preallocated_buffer_size,maximum_buffer_size*}]
[**-cc_scratch_buf** {**yes** | **no**}]
[**-clock_source** {**aix** | **switch**}]
[**-cmdfile** *commands_file*]
[**-coredir** *directory_prefix_string* | **none**]
[**-corefile_format** { *lightweight_corefile_name* | **STDERR** }]
[**-corefile_sigterm** {**yes** | **no**}]
[**-cpu_use** *cpu_specifier*]
[**-css_interrupt** {**yes** | **no**}]
[**-debug_notimeout** *non-null string of characters*]
[**-devtype** {**ib**}][**-eager_limit** *size_limit*]
[**-euidevelop** {**yes** | **no** | **deb** | **min** | **nor**}]
[**-euidevice** *device_specifier*]
[**-euilib** {*ip* | *us*}]
[**-euilibpath** *path_specifier*]
[**-hints_filtered** {**yes** | **no**}]
[{**-hostfile** | **-hfile**} *host_file_name*]
[{**-infolevel** | **-ilevel**} *message_level*]
[**-io_buffer_size** *buffer_size*]
[**-io_errlog** {**yes** | **no**}]
[**-ionodefile** *io_node_file_name*]
[**-instances** *number_of_instances*]
[**-labelio** {**yes** | **no**}]
[**-llfile** *loadleveler_job_command_file_name*]
[**-msg_api** {**MPI** | **LAPI** | **MPI_LAPI** |**MPI, LAPI** | **LAPI, MPI** }]
[**-msg_envelope_buf** *envelope_buffer_size*]
[**-newjob** {**yes** | **no**}]
[**-nodes** *number_of_nodes*]
[**-pgmmodel** {**spmd** | **mpmd**}]
[**-pmdlog** {**yes** | **no**}]
[**-pmdlog_dir** *directory_name*][**-polling_interval** *interval*]
[**-printenv** {**yes** | **no** | *script_name* }]
[**-procs** *partition_size*]
[**-profdir** *directory_name*]
[**-priority_log** {**yes** | **no**}]
[**-priority_log_dir** *directory_name*]
[**-priority_log_name** *log_file_name*]
[**-priority_ntp** {**yes** | **no**}]
[**-pulse** *interval*]
[**-rc_max_qp** *number_of_queue_pairs*]
[**-rc_use_lmc** {**yes** | **no**}]
[**-rdma_count** {*rCxt block value*| *MPI rCxt block value, LAPI rCxt block value*}]

[**-resd** {**yes** | **no**}]
[**-retransmit_interval** *interval*]
[**-retry** *retry_interval*|**wait**]
[**-retrycount** *retry_count*]
[**-rexmit_buf_cnt** *number of buffers*]
[**-rexmit_buf_size** *buffer_size*]
[**-rmpool** *pool_ID*]
[**-savehostfile** *output_file_name*]
[**-save_llfile** *output_file_name*]
[**-shared_memory** {**yes** | **no**}]
[**-single_thread** {**no** | **yes**}]
[**-statistics** {**yes** | **no**| **print**}]
[**-stdinmode** {**all** | **none** | *task_ID*}]
[**-stdoutmode** {**unordered** | **ordered** | *task_ID*}]
[**-task_affinity** {**SNI** | **MCM** | **mcm_list**}]
[**-tasks_per_node** *number_of_tasks per node*]
[**-thread_stacksize** *stacksize*]
[**-tlp_required** {**none** |**warn** | **kill**}]
[**-udp_packet_size** {*packet_size*}]
[**-use_bulk_xfer** {**yes** | **no**}]
[**-wait_mode** {**nopoll** |**poll** | **sleep** | **yield**}]

The **poe** command invokes the Parallel Operating Environment for loading and executing programs on remote processor nodes. The operation of POE is influenced by a number of POE environment variables. The flag options on this command are each used to temporarily override one of these environment variables. User *program_options* can be freely interspersed with the flag options. If no *program* is specified, POE will either prompt you for programs to load, or, if the **MP_CMDFILE** environment variable is set, will load the partition using the specified commands file.

FLAGS

The **-h** flag, when used, must appear immediately after **poe**, and causes the **poe** man page, if it exists, to be printed to stdout.

Under PE for Linux, the **-v** flag causes a verbose output listing (POE displays the names of all the installed PE and LAPI RPMs). Also see the man page for the **perpms** command.

The remaining flags that you can specify on this command are used to temporarily override POE environment variables. For more information on valid values, and on what a particular flag sets, refer to the description of its associated environment variable in the ENVIRONMENT VARIABLES section. The following flags are grouped by function.

The following Partition Manager control flags override the associated environment variables.

-adapter_use
: **MP_ADAPTER_USE**

-cpu_use
: **MP_CPU_USE**

-devtype
: **MP_DEVTYPE**

-euidevice
  MP_EUIDEVICE

-euilib
  MP_EUILIB

-euilibpath
  MP_EUILIBPATH

-hostfile or -hfile
  MP_HOSTFILE

-procs
  MP_PROCS

-pulse
  MP_PULSE

-rc_max_qp
  MP_RC_MAX_QP

-rc_use_lmc
  MP_RC_USE_LMC

-rdma_count
  MP_RDMA_COUNT

-resd
  MP_RESD

-retry
  MP_RETRY

-retrycount
  MP_RETRYCOUNT

-msg_api
  MP_MSG_API

-rmpool
  MP_RMPOOL

-nodes
  MP_NODES

-tasks_per_node
  MP_TASKS_PER_NODE

-savehostfile
  MP_SAVEHOSTFILE

The following Job Specification flags override the associated environment variables.

-cmdfile
  MP_CMDFILE

-instances
  MP_INSTANCES

-llfile
  MP_LLFILE

-newjob
  MP_NEWJOB

-pgmmodel
: **MP_PGMMODEL**

-save_llfile
: **MP_SAVE_LLFILE**

-task_affinity
: **MP_TASK_AFFINITY**

The following I/O Control flags override the associated environment variables.

-labelio
: **MP_LABELIO**

-stdinmode
: **MP_STDINMODE**

-stdoutmode
: **MP_STDOUTMODE**

The following generation of diagnostic information flags override the associated environment variables.

-infolevel or -ilevel
: **MP_INFOLEVEL**

-pmdlog
: **MP_PMDLOG**

-pmdlog_dir
: **MP_PMDLOG_DIR**

-profdir (PE for Linux only)
: **MP_PROFDIR** (PE for Linux only)

-debug_notimeout
: **MP_DEBUG_NOTIMEOUT**

The following Message Passing flags override the associated environment variables.

-buffer_mem
: **MP_BUFFER_MEM**

-cc_buf_mem
: **MP_CC_BUF_MEM**

-cc_scratch_buf
: **MP_CC_SCRATCH_BUF**

-clock_source
: **MP_CLOCK_SOURCE**

-css_interrupt
: **MP_CSS_INTERRUPT**

-eager_limit
: **MP_EAGER_LIMIT**

-hints_filtered
: **MP_HINTS_FILTERED**

-ionodefile
: **MP_IONODEFILE**

-msg_envelope_buf
MP_MSG_ENVELOPE_BUF

-shared_memory
MP_SHARED_MEMORY

-udp_packet_size
MP_UDP_PACKET_SIZE

-thread_stacksize
MP_THREAD_STACKSIZE

-single_thread
MP_SINGLE_THREAD

-wait_mode
MP_WAIT_MODE

-polling_interval
MP_POLLING_INTERVAL

-retransmit_interval
MP_RETRANSMIT_INTERVAL

-statistics
MP_STATISTICS

-io_buffer_size
MP_IO_BUFFER_SIZE

-io_errlog
MP_IO_ERRLOG

-use_bulk_xfer
MP_USE_BULK_XFER

-bulk_min_msg_size
MP_BULK_MIN_MSG_SIZE

-rexmit_buf_size
MP_REXMIT_BUF_SIZE

-rexmit_buf_cnt
MP_REXMIT_BUF_CNT

The following core file generation flags override the associated environment variables.

-coredir
MP_COREDIR

-corefile_format (PE for AIX only)
MP_COREFILE_FORMAT (PE for AIX only)

-corefile_sigterm (PE for AIX only)
MP_COREFILE_SIGTERM (PE for AIX only)

The following are miscellaneous flags:

-euidevelop
MP_EUIDEVELOP

-printenv
MP_PRINTENV

-statistics
MP_STATISTICS

-priority_log
MP_PRIORITY_LOG

-priority_log_dir
MP_PRIORITY_LOG_DIR

-priority_log_name
MP_PRIORITY_LOG_NAME

-priority_ntp (PE for AIX only)
MP_PRIORITY_NTP (PE for AIX only)

-tlp_required (PE for AIX only)
MP_TLP_REQUIRED (PE for AIX only)

DESCRIPTION

The **poe** command invokes the Parallel Operating Environment for loading and executing programs on remote nodes. You can enter it at your home node to:

- load and execute an SPMD program on all nodes of your partition.
- individually load the nodes of your partition with an MPMD job.
- load and execute a series of SPMD and MPMD programs, in individual job steps, on the same partition.
- run nonparallel programs on remote nodes.

The operation of POE is influenced by a number of POE environment variables. The flag options on this command are each used to temporarily override one of these environment variables. User *program_options* can be freely interspersed with the flag options, and *additional_options* not to be parsed by POE can be placed after a *fence_string* defined by the **MP_FENCE** environment variable. If no *program* is specified, POE will either prompt you for programs to load, or, if the **MP_CMDFILE** environment variable is set, will load the partition using the specified commands file.

The environment variables and flags that influence the operation of this command fall into distinct categories of function. They are:

- **Partition Manager control**. The environment variables and flags in this category determine the method of node allocation, message passing mechanism, and the PULSE monitor function.
- **Job specification**. The environment variables and flags in this category determine whether or not the Partition Manager should maintain the partition for multiple job steps, whether commands should be read from a file or STDIN, and how the partition should be loaded.
- **I/O control**. The environment variables and flags in this category determine how I/O from the parallel tasks should be handled. These environment variables and flags set the input and output modes, and determine whether or not output is labeled by task id.
- **Generation of diagnostic information**. The environment variables and flags in this category enable you to generate diagnostic information that may be required by the IBM Support Center in resolving PE-related problems.
- **Message Passing Interface**. The environment variables and flags in this category enable you to specify values for tuning message passing applications.

- **Corefile generation.** The environment variables and flags in this category govern aspects of core file generation including the directory name into which core files will be saved or, for AIX users, the core file format (standard AIX or lightweight).
- **Miscellaneous**. The additional environment variables and flags in this category enable additional error checking, and set a dispatch priority class for execution.

ENVIRONMENT VARIABLES

The environment variable descriptions in this section are grouped by function.

The following environment variables are associated with Partition Manager control.

MP_ADAPTER_USE
Determines how the node's adapter should be used. The User Space communication subsystem library does not require dedicated use of the IBM High Performance Switch on the node. Adapter use will be defaulted, as in "Step 3b: Create a host list file" on page 22, but shared usage may be specified. Valid values are *dedicated* and *shared*. If not set, the default is dedicated for User Space jobs, or shared for IP jobs. The value of this environment variable can be overridden using the **-adapter_use** flag.

MP_CPU_USE
Determines how the node's CPUs should be used. The User Space communication subsystem library does not require unique CPU use on the node. CPU use will be defaulted, as in "Step 3b: Create a host list file" on page 22, but multiple use may be specified. Valid values are *multiple* and *unique*. If not set, the default is *unique* for User Space jobs, or *multiple* for IP jobs. The value of this environment variable can be overridden using the **-cpu_use** flag.

MP_DEVTYPE
Specifies the device type class. Currently, the only valid value is *ib* (InfiniBand). There is no default value (you must explicitly set **MP_DEVTYPE** to *ib* if you want to use the InfiniBand interconnect).

MP_EUIDEVICE
Determines the adapter set to use for message passing.

Under PE for AIX, valid values are *en0* (for Ethernet), *fi0* (for FDDI), *tr0* (for token-ring), *sn_single* (to specify one User Space window per task), *sn_all* (to specify multiple (striped) User Space windows per task), or a string representing an adapter device name, as configured in LoadLeveler.

Under PE for Linux, valid values are a string representing an adapter device name or network type, configured by LoadLeveler. You can also specify *sn_all* (for one window per task) or *sn_single* (for multiple windows per task).

MP_EUILIB
Determines the communication subsystem implementation to use for communication either the IP communication subsystem or the User Space communication subsystem. In order to use the User Space communication subsystem, you must have a system configured with its IBM High Performance Switch feature. Valid, case-sensitive, values are **ip** (for the IP communication subsystem) or **us** (for the User Space communication subsystem). The value of this environment variable can be overridden using the **-euilib** flag.

MP_EUILIBPATH
Determines the path to the message passing and communication subsystem libraries. This only needs to be set if an alternate library path is desired. Valid values are any path specifier. The value of this environment variable can be overridden using the **-euilibpath** flag.

MP_HOSTFILE
Determines the name of a host list file for node allocation. Valid values are any file specifier. If not set, the default is *host.list* in your current directory. The value of this environment variable can be overridden using the **-hostfile** or **-hfile** flags.

MP_PROCS
Determines the number of program tasks. Valid values are any number from 1 to the maximum supported total task limit. If not set, the default is 1. The value of this environment variable can be overridden using the **-procs** flag.

MP_PULSE
The interval (in seconds) at which POE checks the remote nodes to ensure that they are communicating with the home node. The default interval is 600 seconds (10 minutes). To disable the pulse function, specify an interval of 0 (zero) seconds. You can override the value of this environment variable with the **-pulse** flag.

MP_RC_MAX_QP
Specifies the maximum number of Reliable Connected Queue Pairs (RC QPs) that can be created. The allowable value is any positive integer. The default is 2147483647 (which is unlimited). Note that the purpose of **MP_RC_MAX_QP** is to limit the amount of memory that is consumed by RC QPs. It is suggested that you only set this variable if you suspect that your application is performing poorly due to lack of memory.

MP_RC_USE_LMC
Determines whether LMC (Lid Mask Control) is enabled. Enabling the use of LMC can improve performance, because a single port can support multiple Reliable Connected (RC) paths. The default value is **no** (only one RC connected path is supported). Setting **MP_RC_USE_LMC** to **yes** causes multiple RC paths to be supported, which may improve performance.

MP_ RDMA_COUNT

Note: **MP_RDMA_COUNT** applies to applications that use explicit LAPI RDMA. It never applies for applications that use only MPI. For more information, see *Reliable Scalable Cluster Technology: LAPI Programming Guide*.

Specifies the number of user rCxt blocks. It supports the specification of multiple values when multiple protocols are involved. The format can be one of the following:

- **MP_RDMA_COUNT=m** for a single protocol
- **MP_RDMA_COUNT=m,n** for multiple protocols. Only for when **MP_MSG_API=mpi,lapi** – the values are positional, **m** is for MPI, **n** for LAPI.

Note that the **MP_RDMA_COUNT/–rdma_count** option signifies the number of rCxt blocks the user has requested for the job, and it is up to LoadLeveler to determine the actual number of rCxt blocks that will be allocated for the job. POE uses the value of **MP_RDMA_COUNT** to specify the number of rCxt blocks requested on the LoadLeveler MPI and/or LAPI network information when the job is submitted.

The **MP_RDMA_COUNT** specification only has meaning for LAPI applications that use the LAPI utility operation **LAPI_REMOTE_RCXT**. When **MP_RDMA_COUNT** is specified for MPI applications (either when **MP_MSG_API** is explicitly set or defaults to **mpi**), POE issues a warning message that the **MP_RDMA_COUNT** specification is unnecessary.

MP_REMOTEDIR (PE for AIX only)
Specifies the name of a script which echoes the name of the current directory to be used on the remote nodes. By default, the current directory is the current directory at the time that POE is run. You may need to specify this if the AutoMount Daemon is used to mount user file systems, and the user is not using the Korn shell.

The script **mpamddir** is provided for mapping the C shell directory name to an AutoMount Daemon name.

MP_RESD
Determines whether or not the Partition Manager should connect to LoadLeveler to allocate nodes. Valid values are either **yes** or **no**, and there is no default. The value of this environment variable can be overridden using the **-resd** flag.

MP_RETRY
The period of time (in seconds) between processor node allocation retries by POE if there are not enough processor nodes immediately available to run a program. This is valid only if you are using LoadLeveler. If the (case insensitive) character string **wait** is specified instead of a number, no retries are attempted by POE, and the job remains enqueued in LoadLeveler until LoadLeveler either schedules the job or cancels it.

MP_RETRYCOUNT
The number of times (at the interval set by **MP_RETRY**) that the partition manager should attempt to allocate processor nodes. This value is ignored if **MP_RETRY** is set to the character string **wait**.

MP_MSG_API
To indicate to POE which message-passing API is being used by the parallel tasks. **MPI** indicates to use MPI protocol only. **LAPI** indicates to use **LAPI** protocol only. **MPI_LAPI** indicates that both protocols are used, sharing the same set of communication resources (windows, UDP ports). **MPI, LAPI** indicates that both protocols are used, with dedicated resources assigned to each of them. **LAPI, MPI** has a meaning identical to **MPI, LAPI**.

MP_RMPOOL
Determines the name or number of the pool that should be used for nonspecific node allocation. This environment variable/command line flag only applies to LoadLeveler. Valid values are any identifying pool name or number. There is no default. The value of this environment variable can be overridden using the **-rmpool** flag.

MP_NODES
Specifies the number of physical nodes on which to run the parallel tasks. It may be used alone or in conjunction with **MP_TASKS_PER_NODE** and/or **MP_PROCS**, as described in "Step 3i: Set the MP_RMPOOL environment variable" on page 33. The value of this environment variable can be overridden using the **-nodes** flag.

MP_TASKS_PER_NODE
Specifies the number of tasks to be run on each of the physical nodes. It may be used in conjunction with **MP_NODES** and/or **MP_PROCS**, as described in "Step 3i: Set the MP_RMPOOL environment variable" on page 33, but may not be used alone. The value of this environment variable can be overridden using the **-tasks_per_node** flag.

MP_SAVEHOSTFILE
The name of an output host list file to be generated by the Partition Manager. Valid values are any relative or full path name. The value of this environment variable can be overridden using the **-savehostfile** flag.

MP_TIMEOUT
The length of time, in seconds, that POE waits before abandoning an attempt to connect to the remote nodes. The default is **150**.

MP_CKPTDIR (PE for AIX only)
Defines the directory where the checkpoint files will reside when checkpointing a program. See "Checkpointing and restarting programs (PE for AIX only)" on page 54 for more information.

MP_CKPTFILE (PE for AIX only)
Defines the base name of the checkpoint file when checkpointing a program. See "Checkpointing and restarting programs (PE for AIX only)" on page 54 for more information.

MP_CKPTDIR_PERTASK (PE for AIX only)
Specifies whether the checkpoint files of the parallel tasks should be written to separate subdirectories under the directory that is specified by **MP_CKPTDIR**. The default is **no**.

The subdirectories must exist prior to invoking the parallel checkpoint. Using separate subdirectories may provide better performance when using a shared/parallel file system (for example, GPFS) for checkpointing from more than 128 nodes, depending on the specifics of the file system, checkpoint file size, and other factors. The subdirectory name used for each task is its task number.

The following environment variables are associated with Job Specification.

MP_CMDFILE
Determines the name of a POE commands file used to load the nodes of your partition. If set, POE will read the commands file rather than STDIN. Valid values are any file specifier. The value of this environment variable can be overridden using the **-cmdfile** flag.

MP_INSTANCES
The number of instances of User Space windows or IP addresses to be assigned per task per protocol per network. This value is expressed as an integer, or the string **max**. If the value specified exceeds the maximum allowed number of instances, as determined by LoadLeveler, the true maximum number determined is substituted.

MP_LLFILE
Determines the name of a LoadLeveler job command file for node allocation. If you are performing specific node allocation, you can use a LoadLeveler job command file in conjunction with a host list file. If you do, the specific nodes listed in the host list file will be requested from

LoadLeveler. Valid values are any relative or full path name. The value of this environment variable can be overridden using the **-llfile** environment variable.

MP_NEWJOB
Determines whether or not the Partition Manager maintains your partition for multiple job steps. Valid values are **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-newjob** flag.

MP_PGMMODEL
Determines the programming model you are using. Valid values are **spmd** or **mpmd**. If not set, the default is **spmd**. The value of this environment variable can be overridden using the **-pgmmodel** flag.

MP_SAVE_LLFILE
When using LoadLeveler for node allocation, the name of the output LoadLeveler job command file to be generated by the Partition Manager. The output LoadLeveler job command file will show the LoadLeveler settings that result from the POE environment variables and/or command line options for the current invocation of POE. If you use the **MP_SAVE_LLFILE** environment variable for a batch job, or when the **MP_LLFILE** environment variable is set (indicating that a LoadLeveler job command file should participate in node allocation), POE will show a warning and will not save the output job command file. Valid values are any relative or full path name. The value of this environment variable can be overridden using the **-save_llfile** flag.

MP_TASK_AFFINITY
Setting this environment variable causes the PMD to attach each task of a parallel job to one of the system rsets (or AIX) or cpusets (for Linux) at the MCM level. This constrains the task, and all its threads, to run within that MCM. If the task has an inherited rset or cpuset, the attach honors the constraints of the inherited set. When POE is run under LoadLeveler (which includes all User Space jobs), POE relies on LoadLeveler to handle scheduling affinity, based on LoadLeveler job control file keywords that POE sets up in submitting the job. Memory and task affinity must be enabled in the LoadLeveler configuration file (using the **RSET_SUPPORT** keyword).

For more information, about handling task affinity, see “Managing task affinity on large SMP nodes” on page 57.

With interactive POE jobs, the possible **MP_TASK_AFFINITY** values are shown below. Note that these values are not case sensitive. For more specific details about the value that can be provided with **MP_TASK_AFFINITY**, see “Managing task affinity on large SMP nodes” on page 57.

- **MCM** – Specifies that the tasks are allocated in a round-robin fashion among the MCM’s attached to the job by WLM. By default, the tasks are allocated to all the MCMs in the node.
- **SNI** – Specifies that the tasks are allocated to the MCM in common with the first adapter assigned to the task by LoadLeveler. This applies only to User Space MPI jobs. **MP_TASK_AFFINITY=SNI** should not be specified for IP jobs.
- **CORE** – Specifies that each MPI task runs on a single physical processor core. If simultaneous multithreading (SMT) is disabled, this may be one CPU. If SMT is enabled, it may be two CPUs.

- **CPU –** Specifies that each MPI task runs on a single logical CPU.
- **CORE:*n* –** Specifies the number of processor cores to which the threads of an MPI task are constrained (one thread per core), where ***n*** is an integer between 1 and 99999. When you specify the value for ***n*,** you must precede it with a colon (**:*n***). This option signifies the use of OpenMP support, which requires XL Fortran Version 11 PTF1 (or later) or XL C/C++ Version 9.0 (or later).

  This option requires LoadLeveler 3.4.3, or later (when an interactive POE job is run under LoadLeveler).
- **CPU:*n* –** Specifies the number of logical CPUs to which the threads of an MPI task are constrained (one thread per CPU), where ***n*** is an integer between 1 and 99999. When you specify the value for ***n*,** you must precede it with a colon (**:*n***). This option signifies the use of OpenMP support, which requires XL Fortran Version 11 PTF1 (or later) or XL C/C++ Version 9.0 (or later).

  This option requires LoadLeveler 3.4.3 , or later (when an interactive POE job is run under LoadLeveler).
- **mcm-list –** Specifies that the tasks are assigned on a round-robin basis to this set, within the constraint of an inherited rset, if any. **mcm-list** specifies a set of system level (LPAR) logical MCMs to which tasks can be attached. Assignments to MCMs that are outside the constraint set are attempted, but will fail. If a single MCM number is specified as the list, all tasks are assigned to that MCM. If a single MCM number is specified as the list, all tasks are assigned to that MCM. This option is only valid when running either without LoadLeveler, or with LoadLeveler Version 3.2 (or earlier), which does not support scheduling affinity.
- **-1 –** Specifies that no affinity request will be made (disables task affinity).

The following environment variables are associated with STDIO Control.

MP_LABELIO
Determines whether or not output from the parallel tasks are labeled by task id. Valid values are **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-labelio** flag.

MP_STDINMODE
Determines the input mode how STDIN is managed for the parallel tasks. Valid values are:

all all tasks receive the same input data from STDIN.

none no tasks receive input data from STDIN; STDIN will be used by the home node only.

n STDIN is only sent to the task identified (*n*).

If not set, the default is **all**. The value of this environment variable can be overridden using the **-stdinmode** flag.

MP_STDOUTMODE
Determines the output mode how STDOUT is handled by the parallel tasks. Valid values are:

unordered
all tasks write output data to STDOUT asynchronously.

ordered
: output data from each parallel task is written to its own buffer. Later, all buffers are flushed, in task order, to STDOUT.

a task id
: only the task indicated writes output data to STDOUT.

If not set, the default is **unordered**. The value of this environment variable can be overridden using the **-stdoutmode** flag.

MP_USE_MC
Used to determine whether to leverage the hardware multicast function or use the peer-to-peer method to simulate the multicast function. In IP mode, setting **MP_USE_MC** to **yes** indicates that the IP hardware multicast function will be used. Setting **MP_USE_MC** to **no** indicates that the peer-to-peer method will be used. With User Space protocol, only the peer-to-peer method is currently supported.

Valid values are **yes** and **no**. The default value is **no**.

MP_MC_INET_BASE_ ADDR
Used to determine the base IPv4 multicast address for the current LAPI job.

Valid values include any IPv4 multicast address. The default value is **224.3.0.1**.

MP_MC_INET_ADDR_ LEN
Used to determine the length of the available multicast address range, starting from value specified for **MP_MC_INET_BASE_ADDR**.

Valid values include any integer, 1 or greater, that does not exceed the range of multicast addresses that the operating system permits. The default value is **16**.

MP_MC_INET_PORT
Used to determine the port number used for multicasting.

Valid values include any integer between **1024** and **65535**. The default value is **2008**.

The following environment variables are associated with the generation of diagnostic information.

MP_INFOLEVEL
Determines the level of message reporting. Valid values are:

0	error
1	warning and error
2	informational, warning, and error
3	informational, warning, and error. Also reports diagnostic messages for use by the IBM Support Center.
4, 5, 6	informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center.

If not set, the default is **1** (warning and error). The value of this environment variable can be overridden using the **-infolevel** or **-ilevel** flag.

MP_PMDLOG
Determines whether or not diagnostic messages should be logged to a file

in */tmp* on each of the remote nodes. Typically, this environment variable/command line flag is only used under the direction of the IBM Support Center in resolving a PE-related problem. Valid values are **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-pmdlog** flag.

MP_PMDLOG_DIR
Specifies the directory in which the diagnostic log file, generated by setting **MP_PMDLOG** to **yes**, is stored. If **MP_PMDLOG_DIR** is not set, the location of the directory defaults to **/tmp**. If the specified directory is invalid, or you do not have write permission to the specified directory, pmd logging will be disabled and no log file will be created. The value of this environment variable can be overridden using the **-pmdlog_dir** flag.

MP_PRINTENV
Use this environment variable to activate generating a report on the parallel environment setup for the MPI job at hand. The report is printed to STDOUT. The printing of this report will have no adverse effect on the performance of the MPI program. The value can also be a user-specified script name, the output of which will be added to end of the normal environment setup report.

The allowable values for **MP_PRINTENV** are:

no Do not produce a report of environment variable settings. This is the default value.

yes Produce a report of MPI environment variable settings. This report is generated when MPI job initialization is complete.

script_name
Produce the report (same as **yes**), then append the output of the script specified here.

MP_PROFDIR (PE for Linux only)
Allows you to specify the directory into which POE stores the **gmon.out** file for each task. A **gmon.out** file contains profiling data and is produced by compiling a program with the **-pg** flag.

You can specify any string with **MP_PROFDIR**. If you specify a directory name without a leading slash, the new directory is created relative to the working directory. If you include a leading slash, the new directory is created with the absolute path.

If not set, the default directory is **./profdir.***task_id*.

MP_STATISTICS
Provides the ability to gather **MPI** and LAPI communication statistics for MPI user space jobs. Valid values are **yes**, **no** and **print**. If not set, the default is **no** and the values are not case sensitive. The **MPI** statistical information can be used to get a summary on the network usage at the end of the MPI job and to check the progress of inter-job message passing during the execution of an MPI program. To get a summary of the network usage, use **print**. A list of **MPI** statistical information will be printed when MPI_Finalize is called.To check the progress of inter-job message passing, use **yes** and the MPI nonstandard functions 'mpci_statistics_write' and 'mpci_statistics_zero'. The calls must be inserted strategically into the MPI program, and a program that contains them will not be portable to other MPI implementations. The 'mpci_statistics_write' is for printing out the current counters and the 'mpci_statistics_zero' function is for zeroing the counters. These function prototypes are:

```
int  mpci_statistics_zero(void)
int  mpci_statistics_write(FILE  *fptr)
```

Note: Activating MPCI statistics may have a slight impact on performance of the MPI program.

MP_DEBUG_NOTIMEOUT
A debugging aid that allows programmers to attach to one or more of their tasks without the concern that some other task may reach the LAPI timeout. Such a timeout would normally occur if one of the job tasks was continuing to run, and tried to communicate with a task to which the programmer has attached using a debugger. When this flag is set to **yes**, LAPI will never timeout and continue retransmitting message packets forever. The default setting is **no**, allowing LAPI to timeout.

The following environment variables are associated with the Message Passing Interface.

MP_UDP_PACKET_SIZE
Allows the user to control the LAPI UDP datagram size. Specify a positive integer.

MP_ACK_THRESH
Allows the user to control the LAPI packet acknowledgement threshold. Specify a positive integer, no greater than 31. The default is 30.

MP_BUFFER_MEM
Specifies the size of the Early Arrival (EA) buffer that is used by the communication subsystem to buffer eager send message, for point-to-point operations, that arrive before there is a matching receive posted. This value can also be specified with the **-buffer_mem** command line flag. The command line flag will override a value set with the environment variable.

This environment variable can be used in one of two ways:

- Specify the size of a preallocated EA buffer and have PE/MPI guarantee that no valid MPI application can require more EA buffer space than is preallocated. For applications without very large tasks counts or with modest memory demand per task, this form is almost always sufficient.
- Specify the size of a preallocated EA buffer and the maximum size that PE/MPI will guarantee the buffer can never exceed. Aggressive use of EA space is rare in real MPI applications but when task counts are large, the need for PE/MPI to enforce an absolute guarantee may compromise performance. Specifying a preallocated EA buffer that is big enough for the application's real needs but an upper bound that loosens enforcement may provide better performance in some cases, but those cases will not be common.

The default values for preallocated EA space are 64 MB when running with either User Space or IP. (In prior versions of PE for AIX, the preallocation for IP was 2.8 MB which often limited performance. The increase to 64 MB can cause some applications that ran before to fail in a malloc. Such applications can be recompiled with more heap, or can be run by experimenting with **MP_BUFFER_MEM** settings below 64 MB.)

To evaluate whether overriding **MP_BUFFER_MEM** defaults for a particular application is worthwhile, use **MP_STATISTICS**. This tells you whether there is significantly more EA buffer space allocated than is used or whether EA space limits are creating potential performance impacts by forcing some messages that are smaller than the eager limit to use rendezvous protocol because EA buffer cannot be guaranteed.

The **MP_BUFFER_MEM** default value can be defined by the system administrator in the **/etc/poe.limits** file as described in *IBM Parallel Environment: Installation Guide*. If you have not specified **MP_BUFFER_MEM**, and it is set in **/etc/poe.limits**, the default value is set based on the value in **/etc/poe.limits**.

For more information about **MP_BUFFER_MEM** see "Using MP_BUFFER_MEM" on page 67. For information about buffering eager send messages, see *IBM Parallel Environment: MPI Programming Guide*.

MP_CC_BUF_MEM

Specifies the size of the Early Arrival (EA) buffer that is used by the communication subsystem to buffer eager send messages, for collective communications operations, that arrive before there is a matching receive posted. This value can also be specified with the **-cc_buf_mem** command line flag. The command line flag overrides a value set with the environment variable.

This environment variable can be used in one of three ways:

- Specify the size of a preallocated EA buffer and have PE/MPI guarantee that no valid MPI application can require more EA buffer space than is preallocated. For applications without very large task counts or with modest memory demand per task, this form is almost always sufficient.
- Specify the upper bound of a preallocated buffer (the maximum size that PE/MPI will guarantee the buffer can never exceed).
- Specify both the size of a preallocated EA buffer and the maximum size that PE/MPI will guarantee the buffer can never exceed. Aggressive use of EA space is rare in real MPI applications but when task counts are large, the need for PE/MPI to enforce an absolute guarantee may compromise performance. Specifying a preallocated EA buffer that is big enough for the application's real needs but an upper bound that loosens enforcement may provide better performance in some cases, but those cases will not be common.

The default value for preallocated EA space is 4 MB when running with either User Space or IP.

The **MP_CC_BUF_MEM** default value can be defined by the system administrator in the **/etc/poe.limits** file as described in *IBM Parallel Environment: Installation Guide*. If you have not specified **MP_CC_BUF_MEM**, and it is set in **/etc/poe.limits**, the default value is set based on the value in **/etc/poe.limits**.

MP_CC_SCRATCH_BUF

Specifies whether MPI should always use the fastest collective communication algorithm when there are alternatives that require less scratch buffer. In some cases, the faster algorithm needs to allocate more scratch buffers and therefore, consumes more memory than a slower algorithm. The default value is **yes**, which means that you want MPI to choose an algorithm that has the shortest execution time, even though it may consume extra memory. A value of no specifies that MPI should choose the algorithm that uses less memory. Note that restricting MPI to the algorithm that uses the least memory normally sacrifices performance in exchange for that memory savings, so a value of **no** should be specified only when limiting memory usage is critical.

The value of **MP_CC_SCRATCH_BUF** can be overridden with the **-cc_scratch_buf** command line flag.

MP_CLOCK_SOURCE
Under PE for AIX, determines whether or not to use the switch clock as a time source. Valid values are **AIX** and **switch**. There is no default value. The value of this environment variable can be overridden using the **-clock_source** flag.

Under PE for Linux, specifies the time source. Currently, the only valid value is **OS** (operating system).

MP_CSS_INTERRUPT
Determines whether or not arriving message packets cause interrupts. This may provide better performance for certain applications. Valid values are **yes** and **no**. If not set, the default is **no**.

MP_EAGER_LIMIT
Changes the threshold value for message size, above which rendezvous protocol is used.

If the **MP_EAGER_LIMIT** environment variable is not set during initialization, MPI automatically chooses a default eager limit value, based on the number of tasks. For specific information about the default eager limit values, see *IBM Parallel Environment: MPI Programming Guide*.

Consider running a new application once with eager limit set to 0 (zero) because this is useful for confirming that an application is *safe*, but normally higher eager limit gives better performance. Note that a *safe* application, as defined by the MPI standard, is one that does not depend on some minimum of MPI buffer space to avoid deadlock.

The maximum value for **MP_EAGER_LIMIT** is 256K (262144 bytes). Any value that is less than 64 bytes but greater than zero bytes is automatically increased to 64 bytes. A non-power of 2 value will be rounded up to the nearest power of 2. A value may be adjusted if the early arrival buffer (**MP_BUFFER_MEM**) size is set too small.

For information about buffering eager send messages and eager limit, see *IBM Parallel Environment: MPI Programming Guide*.

MP_HINTS_FILTERED
Determines whether MPI info objects reject hints (key/value pairs) which are not meaningful to the MPI implementation. In filtered mode, an **MPI_INFO_SET** call which provides a key/value pair that the implementation does not understand will behave as a no-op. A subsequent **MPI_INFO_GET** call will find that the hint does not exist in the info object.

In unfiltered mode, any key/value pair is stored and may be retrieved. Applications that wish to use MPI info objects to cache and retrieve key/value pairs other than those actually understood by the MPI implementation must use unfiltered mode. The option has no effect on the way MPI uses the hints it does understand. In unfiltered mode, there is no way for a program to discover which hints are valid to MPI and which are simply being carried as uninterpreted key/value pairs.

Providing an unrecognized hint is not an error in either mode.

Valid values for this environment variable are **yes** and **no**. If set to **yes**, unrecognized hints are be filtered. If set to **no**, they will not. If this environment variable is not set, the default is **no**. The value of this environment variable can be overridden using the **-hints_filtered** command line flag.

MP_IONODEFILE
The name of a parallel I/O node file — a text file that lists the nodes that should be handling parallel I/O. This enables you to limit the number of nodes that participate in parallel I/O, guarantee that all I/O operations are performed on the same node, and so on. Valid values are any relative or full path name. If not specified, all nodes will participate in parallel I/O operations. The value of this environment variable can be overridden using the **-ionodefile** command line flag.

MP_MSG_ENVELOPE_BUF
Changes the size of the message *envelope buffer*. You can specify any positive number. There is no upper limit, but any value less than 1 MB is ignored. MPI preallocates the message envelope buffer with a default size of 8 MB. The MPI statistics function prints out the message envelope buffer usage which you can use to determine the best envelope buffer size for a particular MPI program.

The envelope buffer is used for storing both send and receive descriptors. An **MPI_Isend** or unmatched **MPI_Irecv** posting creates a descriptor that lives until the **MPI_Wait** completes. When a message arrives and finds no match, an early arrival descriptor is created that lives until a matching receive is posted and that receive completes in an **MPI_Wait**. For any message at the destination, there will be only one descriptor; either the one created at the **MPI_Irecv** call or the one created at the early arrival. The more uncompleted **MPI_Irecv** and **MPI_Isend** operations an application maintains, the higher the envelope buffer requirement. Most applications will have no reason to adjust the size of this buffer.

The value of **MP_MSG_ENVELOPE_BUF** can be overridden with the **-msg_envelope_buf** command line flag.

MP_LAPI_TRACE_LEVEL
Used for debug purposes. Under PE for AIX, **MP_LAPI_TRACE_LEVEL** is used in conjunction with AIX tracing. Levels 0-6 are supported.

MP_SHARED_MEMORY
To specify the use of shared memory (instead of the network) for message passing between tasks running on the same node. The default value is **yes**.

Note: In past releases of PE for AIX, the **MP_SHM_CC** environment variable was used to enable or disable the use of shared memory for certain 64-bit MPI collective communication operations. Beginning with the PE 4.2 release, this environment variable has been removed. If you are using PE for AIX, you should now use **MP_SHARED_MEMORY** to enable shared memory for both collective communication and point-to-point routines. The default setting for **MP_SHARED_MEMORY** is **yes** (enable shared memory).

MP_USE_BULK_XFER
Exploit the IBM High Performance Switch bulk data transfer mechanism. This variable does not have any meaning and is ignored in other environments.

Before you can use **MP_USE_BULK_XFER**, the system administrator must first enable Remote Direct Memory Access (RDMA). For more information, see *IBM Parallel Environment: Installation*.

Valid values are **yes** and **no**. If not set, the default is **no**.

Note that when you use **MP_USE_BULK_XFER**, you also need to consider the value of the **MP_BULK_MIN_MSG_SIZE** environment variable. Messages with data lengths that are greater than the value specified for **MP_BULK_MIN_MSG_SIZE** will use the bulk transfer path, if it is available. See the description of **MP_BULK_MIN_MSG_SIZE** for more information.

MP_BULK_MIN_MSG_SIZE
Set the minimum message length for bulk transfer. Contiguous messages with data lengths greater than or equal to the value you specify for this environment variable will use the bulk transfer path, if it is available. Messages with data lengths that are smaller than the value you specify for this environment variable, or are noncontiguous, will use packet mode transfer.

The valid range of values is from 4096 to 2147483647 (INT_MAX). The size can be expressed in one of the following ways:

- As a number of bytes
- As a number of KB (1024 bytes), using the letter **k** as a suffix
- As a number of MB (1024 * 1024 bytes), using the letter **m** as a suffix
- As a number of GB (1024 * 1024 * 1024 bytes), using the letter **g** as a suffix.

The default value is 153600.

MP_THREAD_STACKSIZE
Determines the additional stacksize allocated for user programs executing on an MPI service thread. If you allocate insufficient space, the program may encounter a SIGSEGV exception.

MP_SINGLE_THREAD
Avoids mutex lock overheads in a single threaded user program. This is an optimization flag, with values of **no** and **yes**. The default value is **no**, which means the potential for multiple user message passing threads is assumed.

Results are undefined if this variable is set to **yes** with multiple application message passing threads in use. To confirm that is it safe to run the application with **MP_SINGLE_THREAD** set to **yes**, run it once with **MP_SINGLE_THREAD** set to **confirm**. The **confirm** option can only warn you if this run has made MPI calls from more than a single thread. If an application is capable of both single-thread and multithread execution, **confirm** cannot warn you of the potential for multiple threads.

If you want to use the PE barrier synchronization register (BSR), you must set **MP_SINGLE_THREAD** to **yes**. For more information about PE's support of the BSR, see *IBM Parallel Environment: MPI Programming Guide*.

Note: MPI-IO, nonstandard MPE_I nonblocking collective communications, and MPI-1SC (MPI One Sided Communication) cannot be used when **MP_SINGLE_THREAD** is set to **yes**. An application that tries to use nonstandard MPE_I nonblocking collective communications, MPI-IO, or MPI-1SC with **MP_SINGLE_THREAD=yes** will be terminated. MPI calls from multiple user threads cannot be detected and will lead to unpredictable results. **MP_SINGLE_THREAD** may help applications that use many small point-to-point messages, but is less likely to help when the norm is larger messages or collective communication.

MP_WAIT_MODE
: To specify how a thread or task behaves when it discovers it is blocked, waiting for a message to arrive.

MP_RETRANSMIT_INTERVAL
: **MP_RETRANSMIT_INTERVAL**=*nnnnn* and its command line equivalent, **-retransmit_interval**=*nnnnn*, control how often the communication subsystem library checks to see if it should retransmit packets that have not been acknowledged. The value *nnnnn* is the number of polling loops between checks. The acceptable range is 1000 to 400000. The default is 10000 for UDP and 400000 for User Space.

MP_IO_BUFFER_SIZE
: Indicates the default size of the data buffer used by MPI-IO agents. For example:

```
export MP_IO_BUFFER_SIZE=16M
```

sets the default size of the MPI-IO data buffer to 16MB. The default value of the environment variable is the number of bytes corresponding to 16 file blocks. This value depends on the block size associated with the file system storing the file. Valid values are any positive size up to 128MB. The size can be expressed as a number of bytes, as a number of KB (1024 bytes), using the letter **k** as a suffix, or as a number of MB (1024 * 1024 bytes), using the letter **m** as a suffix.

MP_IO_ERRLOG
: Indicates whether to turn on error logging for I/O operations. For example:

```
export MP_IO_ERRLOG=yes
```

turns on error logging. When an error occurs, a line of information will be logged into file */tmp/mpi_io_errdump.app_name.userid.taskid*, recording the time the error occurs, the POSIX file system call involved, the file descriptor, and the returned error number.

MP_REXMIT_BUF_SIZE
: The maximum message size which LAPI will store in its local buffers so as to more quickly free up the user buffer containing message data. This size indicates the size of the local buffers LAPI will allocate to store such messages, and will impact memory usage, while potentially improving performance. Messages larger than this size will continue to be transmitted by LAPI; the only difference is that user buffers will not become available for the user to reuse until the message data has been acknowledged as received by the target. The default user message size is 16352 bytes.

MP_REXMIT_BUF_CNT
: The number of buffers that LAPI must allocate for each target job, each buffer being of the size defined by **MP_REXMIT_BUF_SIZE** * **MP_REXMIT_BUF_CNT**. This count indicates the number of in-flight messages that LAPI can store in its local buffers so as to free up the user's message buffers. If there are no more message buffers left, LAPI will still continue transmission of messages; the only difference is that user buffers will not become available for the user to reuse until the message data has been acknowledged as received by the target. The default number of buffers is 128.

The following are core file generation environment variables:

MP_COREDIR
: Creates a separate directory for each task's core file. The value of this environment variable can be overridden using the **-coredir** flag. A value of ″none″ signifies to bypass creating a new directory resulting in core files written to /tmp.

MP_COREFILE_FORMAT (PE for AIX only)
: Determines the format of core files generated when processes terminate abnormally. If not set, POE will generate standard AIX core files. If set to the string *STDERR*, output will go to standard error. If set to any other string, POE will generate a lightweight core file (conforming to the Parallel Tool Consortium's Standardized Lightweight Corefile Format) for each process in your partition. The string you specify is the name you want to assign to each lightweight core file. By default, these lightweight core files will be saved to subdirectories prefixed by the string *coredir* and suffixed by the task id (as in *coredir.0*, *coredir.1*, and so on). You can specify a prefix other than the default *coredir* by setting the **MP_COREDIR** environment variable. The value of this environment variable can be overridden using the **-corefile_format** flag.

MP_COREFILE_SIGTERM (PE for AIX only)
: Determines if POE should generate a core file when a **SIGTERM** signal is received. Valid values are **yes** and **no**. If not set, the default is **no**.

The following are miscellaneous environment variables:

MP_EUIDEVELOP
: Determines whether PE MPI performs less, normal, or more detailed checking during execution. The additional checking is intended for developing applications, and can significantly slow performance. Valid values are **yes** or **no**, **deb** (for *debug*), **nor** (for *normal*), and **min** (for *minimum*). The **min** value shuts off parameter checking for all send and receive operations, and may improve performance, but should be used only with applications that are very well-validated. If not set, the default is **no**. The value of this environment variable can be overridden using the **-euidevelop** flag.

MP_FENCE
: Determines a *fence_string* to be used for separating options you want parsed by POE from those you do not. Valid values are any string, and there is no default. Once set, you can then use the *fence_string* followed by *additional_options* on the **poe** command line. The *additional_options* will not be parsed by POE. This environment variable has no associated command line flag.

MP_NOARGLIST
: Determines whether or not POE ignores the argument list. Valid values are **yes** and **no**. If set to **yes**, POE will not attempt to remove POE command line flags before passing the argument list to the user's program. This environment variable has no associated command line flag.

MP_PRIORITY
: Determines a coscheduler dispatch parameter set for execution. See "Improving application scalability performance (PE for AIX only)" on page 97 for more information on coscheduler parameters. Valid values are any of the dispatch priority classes set up by the system administrator in the file */etc/poe.priority*, or a string of threshold values, as controlled by the */etc/poe.priority* file contents. This environment variable has no associated command line flag.

Note: If your cluster does not have a global time source (for example, an HPS switch), software synchronization of the node clocks (for example, NTP) is required. Otherwise, the high-priority and low-priority windows might not be sufficiently aligned, causing the coscheduler to be ineffective.

MP_PRIORITY_LOG
Determines whether diagnostic messages should be logged to the POE priority adjustment coscheduler log file on each of the remote nodes. The default directory for this log file is in **/tmp**, with a default file name of **pmadjpri.***jobid***.log**, where *jobid* is a unique job identifier. The directory and the name of the log file can be modified using **MP_PRIORITY_LOG_DIR** and **MP_PRIORITY_LOG NAME**.

MP_PRIORITY_LOG should only be used in conjunction with the POE coscheduler **MP_PRIORITY** variable. Valid values are **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-priority_log** flag.

See "POE priority adjustment coscheduler" on page 97 for more information on the POE coscheduler.

MP_PRIORITY_LOG_DIR
Specifies the directory, on each of the remote nodes, into which the POE priority adjustment coscheduler log file is stored. The default directory is **/tmp**. The name of the log file is **pmadjpri.***jobid***.log**, by default, so after issuing this environment variable, the log file is stored in *directory_name***/pmadjpri.***jobid***.log**. If the default directory is invalid, or you do not have write permission to the specified directory, priority adjustment logging will be disabled and no log file will be created. To specify a name other than **pmadjpri.***jobid***.log**, use the **MP_PRIORITY_LOG_NAME** environment variable.

This variable should only be used when the POE coscheduler **MP_PRIORITY_LOG** variable is set to **yes**. See "POE priority adjustment coscheduler" on page 97 for more information on the POE coscheduler.

MP_PRIORITY_LOG_NAME
Specifies the name of the POE priority adjustment coscheduler log file. The default name is **pmadjpri.***jobid***.log**. The directory into which this log file is placed is **/tmp**, by default, so after issuing this environment variable, the log file is stored in **/tmp/***log_file_name*.*jobid***.log**. To specify a directory other than **/tmp**, use the **MP_PRIORITY_LOG_DIR** environment variable.

This variable should only be used when the POE coscheduler **MP_PRIORITY_LOG** variable is set to **yes**. See "POE priority adjustment coscheduler" on page 97 for more information on the POE coscheduler.

If the file name you specified already exists, an ascending number from 1 through 255 is appended between *jobid* and **.log**. For example, if **/tmp/example_file_name.***jobid***.log** already exists, the new file will be named **/tmp/example_file_name.***jobid***.1.log**. If **/tmp/example_file_name.***jobid***.1.log** already exists, the new file will be named **/tmp/example_file_name.***jobid***.2.log**, and so on, until the file name reaches **/tmp/example_file_name.***jobid***.255.log**. If there are already 255 files, priority adjustment logging will be disabled.

MP_PRIORITY_NTP (PE for AIX only)
Determines whether the POE priority adjustment coscheduler will turn NTP off during the priority adjustment period, or leave it running. Valid

values are **yes** or **no**. The value of **no** (which is the default) instructs the POE coscheduler to turn the NTP daemon off (if it was running) and restart NTP later, after the coscheduler completes. Specify a value of **yes** to inform the coscheduler to keep NTP running during the priority adjustment cycles (if NTP was not running, NTP will not be started). If **MP_PRIORITY_NTP** is not set, the default is **no**. The value of this environment variable can be overridden using the **-priority_ntp** flag. See "POE priority adjustment coscheduler" on page 97 for more information about the POE coscheduler.

MP_TLP_REQUIRED (PE for AIX only)
Specifies to POE whether to check to see if jobs being executed have been compiled for large pages, and when it finds a job that was not, the action to take. Using this variable helps avoid system failures, on systems with a high percentage of memory configured as large pages, related to the execution of large memory parallel jobs that were not compiled for large pages. Valid values are **none**, **warn**, and **kill**. When you set **MP_TLP_REQUIRED** to **warn**, POE detects and issues a warning message for any job that was not compiled for large pages. Setting **MP_TLP_REQUIRED** to **kill** causes POE to detect and kill any job that was not compiled for large pages. The default is **none** (POE takes no action).

EXAMPLES

1. Assume the **MP_PGMMODEL** environment variable is set to **spmd**, and **MP_PROCS** is set to *6*. To load and execute the SPMD program *sample* on the six remote nodes of your partition, enter:

   ```
   poe sample
   ```
2. Assume you have an MPMD application consisting of two programs; *master* and *workers*. These programs are designed to run together and communicate via calls to message passing subroutines. The program *master* is designed to run on one processor node. The *workers* program is designed to run as separate tasks on any number of other nodes. The **MP_PGMMODEL** environment variable is set to **mpmd**, and **MP_PROCS** is set to *6*. To individually load the six remote nodes with your MPMD application, enter:

   ```
   poe
   ```
   Once the partition is established, the **poe** command responds with the prompt:

   ```
   0:host1_name>
   ```
   To load the *master* program as task 0 on host1_name, enter:

   ```
   master
   ```
   The **poe** command responds with a prompt for the next node to load. When you have loaded the last node of your partition, the **poe** command displays the message `Partition loaded...` and begins execution.
3. Assume you want to run three SPMD programs; *setup*, *computation*, and *cleanup* – as job steps on the same partition of nodes. The **MP_PGMMODEL** environment variable is set to *spmd*, and **MP_NEWJOB** is set to **yes**. You enter:

   ```
   poe
   ```
   Once the partition is established, the **poe** command responds with the prompt:

   ```
   Enter program name (or quit):
   ```

To load the program *setup*, enter:

```
setup
```

The program setup executes on all nodes of your partition. When execution completes, the **poe** command again prompts you for a program name. Enter the program names in turn. To release the partition, enter:

```
quit
```

4. To check the process status (using the nonparallel command **ps**) for all remote nodes in your partition, enter:

```
poe ps
```

FILES

host.list (Default host list file)

RELATED INFORMATION

Commands: **mpcc_r**(1), **mpcc**(1), **mpCC_r**(1), **mpCC**(1), **mpxlf_r**(1), **mpfort**(1)

poeckpt

Takes a checkpoint of an interactive, non-LoadLeveler POE job. **This command applies to PE for AIX only.**

SYNOPSIS

poeckpt [**-?**] [**-H**] [**-k**] [**-u** *username*] *pid*

FLAGS

-? Provides a short usage message.

-H Provides help information.

-k Specifies that the job is to be terminated after a successful checkpoint.

-u Specifies the owner of the resulting checkpoint file (used only when root invokes the **poeckpt** command). Note that if the specified owner is not the owner of the process being checkpointed, the restart will fail.

pid
The process id of the POE process for the job to be checkpointed.

DESCRIPTION

poeckpt will checkpoint an interactive POE job, ensuring that job is a non-LoadLeveler POE job, running stand-alone. The process id specified corresponds to the POE process id for the job to be checkpointed. If the process specified is not a POE process or if a POE job is running under LoadLeveler, the command will fail. If the terminate option is specified and the POE job cannot be checkpointed, the terminate option is ignored and the POE job continues to run.

The **poeckpt** command will block until the checkpoint operation completes. Interrupting this command by pressing **Ctrl-c** will cause the checkpoint to be aborted.

This command must be run as the user who owns the specified process or as root. When the **-u** flag is specified and the process is being run by root, **poeckpt** will change the ownership of the checkpoint files to the user name specified. The **-u** flag is ignored when **poeckpt** is run by a non-root user.

Return codes are:

0 Indicates success.

-1 Indicates failure. Occurs with error message(s) containing reasons for failure.

Note: For checkpoint failures, the primary errors reported are actual error numbers as documented in */usr/include/sys/errno.h*. The secondary errors provide additional error information and are documented in */usr/include/sys/chkerror.h*. There may also be further error information reported in string format as *error data*.

ENVIRONMENT VARIABLES

This command responds to the following environment variables:

MP_CKPTDIR
Defines the directory where the checkpoint file created by poeckpt will reside. If unset, the default value is the directory from which poeckpt is run. If the value of **MP_CKPTDIR** that is specified in the environment where poeckpt is invoked is not the same as the value of **MP_CKPTDIR** in the environment of the POE job being checkpointed, the checkpoint file of POE may appear in a different directory than the task checkpoint files.

MP_CKPTFILE
Defines the base name of the checkpoint file created by poeckpt. If unset, the default value is poeckpt.<PID>, where PID is the process ID of the POE process being checkpointed. If the value of **MP_CKPTFILE** that is specified in the environment where poeckpt is invoked is not the same as the value of **MP_CKPTFILE** in the environment of the POE job being checkpointed, the base name of the POE checkpoint file may be different than the base name of the task checkpoint files.

poekill

Terminates all remote tasks for a given program.

SYNOPSIS

poe poekill pgm_name [**poe_options**]

or

rsh remote_node poekill pgm_name

poekill is a Korn shell script that searches for the existence of running programs (named **pgm_name**) owned by the user, and terminates them via SIGTERM signals. If run under POE, **poekill** uses the standard POE mechanism for identifying the set of remote nodes; host.list, LoadLeveler, and so on. If run under rsh, **poekill** applies only to the node specified as remote_node.

FLAGS

When run as a POE program, standard POE flags apply.

DESCRIPTION

poekill determines the user id of the user that submitted the command. It then uses the id to obtain a list of active processes, which is filtered by the **pgm_name** argument into a scratch file in **/tmp**. The file is processed by an awk/gawk script that sends a SIGTERM signal (15) to each process in the list, and echoes the action back to the user. The scratch file is then erased, and the script exits with code of 0.

If you do not provide a pgm_name, an error message is printed and the script exits with a code of 1.

The pgm_name can be a substring of the program name.

RELATED INFORMATION

Commands: **rsh**(1), **poe**(1), **kill**(1)

poerestart

Can be used to restart an interactive POE job. **This command applies to PE for AIX only.**

SYNOPSIS

poerestart [**-?**] [**-H**] [**-s**] *file*

FLAGS

-? Provides a short usage message.

-h Provides extended help information.

-s Specifies that the same hosts should be used for the restarted job as were used for the job that was checkpointed.

file The checkpoint file for the POE process.

DESCRIPTION

poerestart will restart a previously checkpointed interactive POE job, from the checkpoint file specified. Only an interactive job, stand-alone or running under LoadLeveler, can be restarted. A batch POE job cannot be restarted with this command.

Interrupting the **poerestart** command by pressing **Ctrl-c** will cause the restart operation to be aborted.

This command must be run as the user who owned the original checkpointed process.

ENVIRONMENT VARIABLES

This command responds to the following environment variables:

MP_HOSTFILE
Specifies the name of the host file to be used. This setting is ignored if the **-s** flag is specified.

MP_RMPOOL
Specifies the name of the LoadLeveler pool from which nodes will be selected to restart the job. It is an error to use this specification if the originally checkpointed POE job was not being run under LoadLeveler. This setting is ignored if:

- The **-s** flag is specified.
- **MP_HOSTFILE** is set.
- A *host.list* file exists in the directory from which the command is run.
- **MP_LLFILE** is set.

MP_LLFILE
Specifies the name of the LoadLeveler job command file to be used for specification of the restarted job. This **must** be specified if the originally checkpointed POE job used the **-llfile** command line option or the **MP_LLFILE** environment variable for job specification. This **cannot** be

used if the originally checkpointed POE job did not use the **-llfile** command line option or the **MP_LLFILE** environment variable for job specification.

NOTES

1. When restarting a non-LoadLeveler job, or a LoadLeveler job that does not use **MP_RMPOOL** or **MP_LLFILE**, the hosts will be determined using the following:
   - The **-s** flag.
   - The **MP_HOSTFILE** environment variable.
   - A *host.list* file.
2. When **MP_LLFILE** is not being used, one of the following **must** be true:
   - The **-s** flag is specified.
   - The **MP_HOSTFILE** environment variable is set.
   - A *host.list* file exists in the directory from which the command is being run.
   - The **MP_RMPOOL** environment variable is set.
3. The following may be used in conjunction with the **MP_LLFILE** environment variable:
   - The **-s** flag.
   - The **MP_HOSTFILE** environment variable.
   - A *host.list* file in the directory from which the command is being run.
4. Any POE environment variables other than those indicated above are not used by the restarted POE.
5. The task geometry (tasks that are common within a node) for the restarted task must be the same as the originally started task.
6. This command may not be used to restart from a checkpoint file of a POE batch job. If the file provided to the **poerestart** command was generated from the checkpoint of a batch POE job, the **poerestart** command will return with no error message printed. The #@error file specified in the original batch job (if present) will contain a message indicating that this error occurred.

Return codes are:

0	Indicates success.
-1	Indicates failure. Occurs with error message(s) containing reasons for failure.

rset_query

When run under POE, can be used to verify that memory affinity assignments are performed. **This command applies to PE for AIX only.**

SYNOPSIS

rset_query

DESCRIPTION

rset_query is used to verify that memory affinity assignments are performed, as an extension of the POE and LoadLeveler scheduling affinity functions. For more information, see "Managing task affinity on large SMP nodes" on page 57.

rset_query does not require any input or arguments. Output is written to STDERR.

EXAMPLES

1. To verify that memory affinity assignments have been performed, run the **rset_query** command under POE, using task affinity. For example, to show affinity assignments for two tasks, each on a different node, issue the command shown below. Note that this example assumes you are using a host list file that lists the two nodes on the first two lines.

   ```
   poe rset_query -procs 2 -labelio yes -euilib us -task_affinity mcm
   ```

   You will see output similar to this:

   ```
   0:ra_getrset returned RS_PARTITION_RSET
   0:Number of available processors: 16
   0:Number of available memory pools: 0
   0:Amount of available memory: 0 MB
   0:Maximum system detail level: 6
   0:SMP detail level: 2
   0:MCM detail level: 3
   0:  Processor  0 in resource set
   0:  Processor  1 in resource set
   0:  Processor  2 in resource set
   0:  Processor  3 in resource set
   0:  Processor  4 in resource set
   0:  Processor  5 in resource set
   0:  Processor  6 in resource set
   0:  Processor  7 in resource set
   0:  Processor  8 in resource set
   0:  Processor  9 in resource set
   0:  Processor 10 in resource set
   0:  Processor 11 in resource set
   0:  Processor 12 in resource set
   0:  Processor 13 in resource set
   0:  Processor 14 in resource set
   0:  Processor 15 in resource set
   0:numrads = 1
   0:
   0:MCM detail:
   0:MCM 0 found:
   0:Number of available processors: 16
   0:Number of available memory pools: 0
   0:Amount of available memory: 0 MB
   0:Maximum system detail level: 6
   0:SMP detail level: 2
   0:MCM detail level: 3
   ```

```
0:  Processor  0 in resource set
0:  Processor  1 in resource set
0:  Processor  2 in resource set
0:  Processor  3 in resource set
0:  Processor  4 in resource set
0:  Processor  5 in resource set
0:  Processor  6 in resource set
0:  Processor  7 in resource set
0:  Processor  8 in resource set
0:  Processor  9 in resource set
0:  Processor 10 in resource set
0:  Processor 11 in resource set
0:  Processor 12 in resource set
0:  Processor 13 in resource set
0:  Processor 14 in resource set
0:  Processor 15 in resource set
1:ra_getrset returned RS_PARTITION_RSET
1:Number of available processors: 16
1:Number of available memory pools: 0
1:Amount of available memory: 0 MB
1:Maximum system detail level: 6
1:SMP detail level: 2
1:MCM detail level: 3
1:  Processor  0 in resource set
1:  Processor  1 in resource set
1:  Processor  2 in resource set
1:  Processor  3 in resource set
1:  Processor  4 in resource set
1:  Processor  5 in resource set
1:  Processor  6 in resource set
1:  Processor  7 in resource set
1:  Processor  8 in resource set
1:  Processor  9 in resource set
1:  Processor 10 in resource set
1:  Processor 11 in resource set
1:  Processor 12 in resource set
1:  Processor 13 in resource set
1:  Processor 14 in resource set
1:  Processor 15 in resource set
1:numrads = 1
1:
1:MCM detail:
1:MCM 0 found:
1:Number of available processors: 16
1:Number of available memory pools: 0
1:Amount of available memory: 0 MB
1:Maximum system detail level: 6
1:SMP detail level: 2
1:MCM detail level: 3
1:  Processor  0 in resource set
1:  Processor  1 in resource set
1:  Processor  2 in resource set
1:  Processor  3 in resource set
1:  Processor  4 in resource set
1:  Processor  5 in resource set
1:  Processor  6 in resource set
1:  Processor  7 in resource set
1:  Processor  8 in resource set
1:  Processor  9 in resource set
1:  Processor 10 in resource set
1:  Processor 11 in resource set
1:  Processor 12 in resource set
1:  Processor 13 in resource set
1:  Processor 14 in resource set
1:  Processor 15 in resource set
```

Chapter 7. POE Environment variables and command line flags

PE includes a number of environment variables and command line flags you can use to influence the execution of parallel programs and the operation of certain tools. The command line flags temporarily override their associated environment variables.

The environment variables and command line flags shown here are divided into tables, depending on the PE function to which they relate.

- Table 54 on page 216 summarizes the environment variables and flags for controlling the Partition Manager. These environment variables and flags enable you to specify such things as an input or output host list file, and the method of node allocation. For a complete description of the variables and flags summarized in this table, see Chapter 2, “Executing parallel programs,” on page 11.
- Table 55 on page 221 summarizes the environment variables and flags for Job Specifications. These environment variables and flags determine whether or not the Partition Manager should maintain the partition for multiple job steps, whether commands should be read from a file or STDIN, and how the partition should be loaded. For a complete description of the variables and flags summarized in this table, see Chapter 2, “Executing parallel programs,” on page 11.
- Table 56 on page 223 summarizes the environment variables and flags for determining how I/O from the parallel tasks should be handled. These environment variables and flags set the input and output modes, and determine whether or not output is labeled by task id. For a complete description of the variables and flags summarized in this table, see “Managing standard input, output, and error” on page 45.
- Table 57 on page 225 summarizes the environment variables and flags for collecting diagnostic information. These environment variables and flags enable you to generate diagnostic information that may be required by the IBM Support Center in resolving PE-related problems.
- Table 58 on page 226 summarizes the environment variables and flags for the Message Passing Interface. These environment variables and flags allow you to change message and memory sizes, as well as other message passing information.
- Table 59 on page 234 summarizes the variables and flags for core file generation (PE for AIX only).
- Table 60 on page 235 summarizes some miscellaneous environment variables and flags. These environment variables and flags enable additional error checking and let you set a dispatch priority class for execution.

You can use the POE command line flags on the **poe** command. You can also use the following flags on program names when individually loading nodes from STDIN or a POE commands file.

- **-infolevel** or **-ilevel**
- **-euidevelop**

Table 54 summarizes the environment variables and flags for controlling the Partition Manager. It includes information about how to set each variable, the values that may be specified, and the default value. These environment variables and flags enable you to specify such things as an input or output host list file, and the method of node allocation. For a complete description of the variables and flags summarized in this table, see Chapter 2, "Executing parallel programs," on page 11.

Table 54. POE environment variables and command line flags for partition manager control

The Environment Variable/Command Line Flag(s):	**Set:**	**Possible Values:**	**Default:**
MP_ADAPTER_USE -adapter_use	How the node's adapter should be used. The User Space communication subsystem library does not require dedicated use of the IBM High Performance Switch on the node. Adapter use will be defaulted, as in "Step 3b: Create a host list file" on page 22, but shared usage may be specified.	One of the following strings: **dedicated** Only a single program task can use the adapter. **shared** A number of tasks on the node can use the adapter.	**dedicated** (for User Space jobs) **shared** (for IP jobs)
MP_CPU_USE -cpu_use	How the node's CPU should be used. The User Space communication subsystem library does not require unique CPU use on the node. CPU use will be defaulted, as in "Step 3b: Create a host list file" on page 22, but multiple use may be specified. For example, either one job per node gets all CPUs, or more than one job can go on a node.	One of the following strings: **unique** Only your program's tasks can use the CPU. **multiple** Your program may share the node with other users.	**unique** (for User Space jobs) **multiple** (for IP jobs) For more details on the default values for MP_CPU_USE, see Table 11 on page 25.
MP_DEVTYPE -devtype	Specifies the device type class. Note that you can only specify a single device type per parallel job; device types may not be mixed.	One of the following strings: **ib** InfiniBand	None

Table 54. POE environment variables and command line flags for partition manager control (continued)

The Environment Variable/Command Line Flag(s):	Set:	Possible Values:	Default:
MP_EUIDEVICE -euidevice	The adapter set to use for message passing.	For AIX, one of the following: **en0** Ethernet **fi0** FDDI **tr0** token-ring **sn_all** Multiple US windows per task **sn_single** One US window per task **ml0** *adapter device name or network type string* Adapter device name or network type string as configured in LoadLeveler. For Linux, one of the following: **sn_all** Multiple windows per task **sn_single** One window per task **eth***x* Gigabit Ethernet (GigE) *adapter device name or network type string* Adapter device name or network type string as configured in LoadLeveler.	The adapter set used as the external network address.
MP_EUILIB -euilib	The communication subsystem implementation to use for communication – either the IP communication subsystem or the User Space communication subsystem.	One of the following strings: **ip** The IP communication subsystem. **us** The User Space communication subsystem. **Note:** This specification is case-sensitive.	**ip**

Table 54. POE environment variables and command line flags for partition manager control (continued)

The Environment Variable/Command Line Flag(s):	Set:	Possible Values:	Default:
MP_EUILIBPATH -euilibpath	The path to the message passing and communication subsystem libraries. This only needs to be set if the libraries are moved, or an alternate set is being used.	Any path specifier.	**/usr/lpp/ppe.poe/lib** (AIX) or **/opt/ibmhpc/ppe.poe/lib/libmpi** (Linux)
MP_HOSTFILE -hostfile -hfile	The name of a host list file for node allocation.	Any file specifier or the word NULL.	**host.list** in the current directory.
MP_INSTANCES -instances	The number of instances of User Space windows or IP addresses to be assigned. This value is expressed as an integer, or the string **max**. If the values specified exceeds the maximum allowed number of instances, as determined by LoadLeveler, that number is substituted.	A positive integer, or the string **max**.	1
MP_PROCS -procs	The number of program tasks.	Any number from 1 to the maximum supported configuration.	1
MP_PULSE -pulse	The interval (in seconds) at which POE checks the remote nodes to ensure that they are actively communicating with the home node.	An integer greater than or equal to 0.	**600**
MP_RESD -resd	Whether or not the Partition Manager should connect to LoadLeveler to allocate nodes. **Note:** When running POE from a workstation that is external to the LoadLeveler cluster, the *LoadL.so* file set must be installed on the external node (see *Tivoli Workload Scheduler LoadLeveler: Using and Administering* and *IBM Parallel Environment: Installation* for more information).	**yes** **no**	Context dependent
MP_RETRY -retry	The period of time (in seconds) between processor node allocation retries by POE if there are not enough processor nodes immediately available to run a program. This is valid only if you are using LoadLeveler. If the character string **wait** is specified instead of a number, no retries are attempted by POE, and the job remains enqueued in LoadLeveler until LoadLeveler either schedules the job or cancels it.	An integer greater than or equal to 0, or the case-insensitive value **wait**.	**0** (no retry)

Table 54. POE environment variables and command line flags for partition manager control (continued)

The Environment Variable/Command Line Flag(s):	Set:	Possible Values:	Default:
MP_RETRYCOUNT -retrycount	The number of times (at the interval set by **MP_RETRY**) that the partition manager should attempt to allocate processor nodes. This value is ignored if **MP_RETRY** is set to the character string **wait**.	An integer greater than or equal to 0.	**0**
MP_MSG_API -msg_api	To indicate to POE which message passing API is being used by the application code. **MPI** Indicates that the application makes only MPI calls. **LAPI** Indicates that the application makes only LAPI calls. **MPI_LAPI** Indicates that calls to both message passing APIs are used in the application, and the same set of communication resources (windows, IP addresses) is to be shared between them. **MPI,LAPI** Indicates that calls to both message passing APIs are used in the application, with dedicated resources assigned to each of them. **LAPI,MPI** Has a meaning identical to **MPI,LAPI**.	**MPI** **LAPI** **MPI_LAPI** **MPI,LAPI** **LAPI,MPI**	**MPI**
MP_RMPOOL -rmpool	The name or number of the pool that should be used for nonspecific node allocation. This environment variable/command line flag only applies to LoadLeveler.	An identifying pool name or number.	None
MP_NODES -nodes	To specify the number of processor nodes on which to run the parallel tasks. It may be used alone or in conjunction with **MP_TASKS_PER_NODE** and/or **MP_PROCS**, as described in "Step 3i: Set the MP_RMPOOL environment variable" on page 33.	Any number from 1 to the maximum supported configuration.	None
MP_TASKS_PER_ NODE -tasks_per_node	To specify the number of tasks to be run on each of the physical nodes. It may be used in conjunction with **MP_NODES** and/or **MP_PROCS**, as described in "Step 3i: Set the MP_RMPOOL environment variable" on page 33, but may not be used alone.	Any number from 1 to the maximum supported configuration.	None

Table 54. POE environment variables and command line flags for partition manager control (continued)

The Environment Variable/Command Line Flag(s):	**Set:**	**Possible Values:**	**Default:**
MP_SAVEHOSTFILE -savehostfile	The name of an output host list file to be generated by the Partition Manager.	Any relative or full path name.	None
MP_REMOTEDIR (no associated command line flag) **(Applies to PE for AIX only)**	The name of a script which echoes the name of the current directory to be used on the remote nodes.	Any file specifier.	None
MP_TIMEOUT (no associated command line flag)	The length of time, in seconds, that POE waits before abandoning an attempt to connect to the remote nodes.	Any number greater than 0. If set to 0 or a negative number, the value is ignored.	**150**
MP_CKPTFILE (no associated command line flag) **(Applies to PE for AIX only)**	The base name of the checkpoint file.	Any file specifier.	See "Checkpointing and restarting programs (PE for AIX only)" on page 54
MP_CKPTDIR (no associated command line flag) **(Applies to PE for AIX only)**	The directory where the checkpoint files will reside.	Any path specifier.	Directory from which POE is run.
MP_CKPTDIR_PERTASK (no associated command line flag) **(Applies to PE for AIX only)**	Specifies whether the checkpoint files of the parallel tasks should be written to separate subdirectories under the directory that is specified by **MP_CKPTDIR**.	**yes** **no**	**no**

Table 55 on page 221 summarizes the environment variables and flags for Job Specification. It includes information about how to set each variable, the values that may be specified, and the default value. These environment variables and flags determine whether or not the Partition Manager should maintain the partition for multiple job steps, whether commands should be read from a file or STDIN, and how the partition should be loaded. For a complete description of the variables and flags summarized in this table, see Chapter 2, "Executing parallel programs," on page 11.

Table 55. POE environment variables and command line flags for job specification

The Environment Variable/Command Line Flag(s):	Set:	Possible Values:	Default:
MP_CMDFILE -cmdfile	The name of a POE commands file used to load the nodes of your partition. If set, POE will read the commands file rather than STDIN.	Any file specifier.	None
MP_LLFILE -llfile	The name of a LoadLeveler job command file for node allocation. If you are performing specific node allocation, you can use a LoadLeveler job command file in conjunction with a host list file. If you do, the specific nodes listed in the host list file will be requested from LoadLeveler.	Any path specifier.	None
MP_NEWJOB -newjob	Whether or not the Partition Manager maintains your partition for multiple job steps.	**yes** **no**	**no**
MP_PGMMODEL -pgmmodel	The programming model you are using.	**spmd** **mpmd**	**spmd**
MP_SAVE_LLFILE -save_llfile	When using LoadLeveler for node allocation, the name of the output LoadLeveler job command file to be generated by the Partition Manager. The output LoadLeveler job command file will show the LoadLeveler settings that result from the POE environment variables and/or command line options for the current invocation of POE. If you use the **MP_SAVE_LLFILE** environment variable for a batch job, or when the **MP_LLFILE** environment variable is set (indicating that a LoadLeveler job command file should participate in node allocation), POE will show a warning and will not save the output job command file.	Any relative or full path name.	None

Table 55. POE environment variables and command line flags for job specification (continued)

The Environment Variable/Command Line Flag(s):	Set:	Possible Values:	Default:
MP_TASK_AFFINITY -task_affinity	Setting this environment variable causes the PMD to attach each task of a parallel job to one of the system rsets (for AIX) or cpusets (for Linux) at the MCM level. This constrains the task and all of its threads to run within that MCM. If the task has an inherited rset or cpuset, the attach honors the constraints of the inherited set. When POE is run under LoadLeveler (which includes all User Space jobs), POE relies on LoadLeveler to handle scheduling affinity, based on LoadLeveler job control file keywords that POE sets up in submitting the job. Memory and task affinity must be enabled in the LoadLeveler configuration file (using the **RSET_SUPPORT** keyword). For more information about handling task affinity and the values that can be specified with the **MP_TASK_AFFINITY** environment variable, see "Managing task affinity on large SMP nodes" on page 57.	**MCM** Specifies that the tasks are allocated in a round-robin fashion among the MCM's attached to the job by WLM. **SNI** Specifies that the tasks are allocated to the MCM in common with the first adapter assigned to the task by LoadLeveler. **CORE** Specifies that each MPI task runs on a single physical processor core. **CPU** Specifies that each MPI task runs on a single logical CPU. **CORE:n** Specifies the number of processor cores to which the threads of an MPI task are constrained (one thread per core), where ***n*** is an integer between 1 and 99999. **CPU:n** Specifies the number of logical CPUs to which the threads of an MPI task are constrained (one thread per CPU), where ***n*** is an integer between 1 and 99999. **mcm-list** Specifies that the tasks are assigned on a round-robin basis to this set, within the constraint of an inherited rset, if any. **-1** Specifies that no affinity request will be made (disables task affinity).	None

Table 56 on page 223 summarizes the environment variables and flags for determining how I/O from the parallel tasks should be handled. It includes information about how to set each variable, the values that may be specified, and the default value. These environment variables and flags set the input and output modes, and determine whether or not output is labeled by task id. For a complete description of the variables and flags summarized in this table, see “Managing standard input, output, and error” on page 45.

Table 56. POE environment variables and command line flags for I/O control

The Environment Variable/Command Line Flag(s):	**Set:**	**Possible Values:**	**Default:**
MP_LABELIO -labelio	Whether or not output from the parallel tasks is labeled by task id.	**yes** **no**	**no**
MP_STDINMODE -stdinmode	The input mode. This determines how input is managed for the parallel tasks.	**all** All tasks receive the same input data from STDIN. **none** No tasks receive input data from STDIN; STDIN will be used by the home node only. **a task id** STDIN is only sent to the task identified.	**all**
MP_STDOUTMODE -stdoutmode	The output mode. This determines how STDOUT is handled by the parallel tasks.	One of the following: **unordered** All tasks write output data to STDOUT asynchronously. **ordered** Output data from each parallel task is written to its own buffer. Later, all buffers are flushed, in task order, to STDOUT. *a task id* Only the task indicated writes output data to STDOUT.	**unordered**

Table 56. POE environment variables and command line flags for I/O control (continued)

The Environment Variable/Command Line Flag(s):	**Set:**	**Possible Values:**	**Default:**
MP_USE_MC	Used to determine whether to leverage the hardware multicast function or use the peer-to-peer method to simulate the multicast function. In IP mode, setting **MP_USE_MC** to **yes** indicates that the IP hardware multicast function will be used. Setting **MP_USE_MC** to **no** indicates that the peer-to-peer method will be used. With User Space protocol, only the peer-to-peer method is currently supported.	**yes** **no**	**no**
MP_MC_INET_BASE_ ADDR	Used to determine the base IPv4 multicast address for the current LAPI job.	Any IPv4 multicast address	**224.3.0.1**
MP_MC_INET_ADDR_ LEN	Used to determine the length of the available multicast address range, starting from the value specified for **MP_MC_INET_BASE_ADDR**.	Any integer, 1 or greater, that does not exceed the range of multicast addresses that the operating system permits.	**16**
MP_MC_INET_PORT	Used to determine the port number used for multicasting.	**1024 - 65535**	**2008**

Table 57 on page 225 summarizes the environment variables and flags for collecting diagnostic information. It includes information about how to set each variable, the values that may be specified, and the default value. These environment variables and flags enable you to generate diagnostic information that may be required by the IBM Support Center in resolving PE-related problems.

Table 57. POE environment variables and command line flags for diagnostic information

The Environment Variable/Command Line Flag(s):	Set:	Possible Values:	Default:
MP_INFOLEVEL -infolevel and -ilevel	The level of message reporting. You can use the **infolevel** and **ilevel** flags when individually loading nodes. For more information on individually loading nodes, refer to “Invoking an MPMD program” on page 36.	One of the following integers: **0** Error **1** Warning and error **2** Informational, warning, and error **3** Informational, warning, and error. Also reports high-level diagnostic messages for use by the IBM Support Center. **4, 5, 6** Informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center.	**1**
MP_PMDLOG -pmdlog	Whether or not diagnostic messages should be logged to a file in **/tmp** on each of the remote nodes. Typically, this environment variable/command line flag is only used under the direction of the IBM Support Center in resolving a PE-related problem.	**yes** **no**	**no**
MP_PMDLOG_DIR -pmdlog_dir	The directory in which the diagnostic log file, generated by setting **MP_PMDLOG** to **yes**, is stored.	Any relative path name or full path name.	**/tmp**
MP_DEBUG_ NOTIMEOUT -debug_notimeout	A debugging aid that allows programmers to attach to one or more of their tasks without the concern that some other task may reach a timeout.	**yes** **no**	**no**

Table 57. POE environment variables and command line flags for diagnostic information (continued)

The Environment Variable/Command Line Flag(s):	Set:	Possible Values:	Default:
MP_PROFDIR -profdir **(Applies to PE for Linux only)**	Allows you to specify the directory into which POE stores the **gmon.out** file for each task. A **gmon.out** file contains profiling data and is produced by compiling a program with the **-pg** flag. You can specify any string with **MP_PROFDIR**. If you specify a directory name without a leading slash, the new directory is created relative to the working directory. If you include a leading slash, the new directory is created with the absolute path. If not set, the default directory is **/profdir.***task_id*.	Any relative path name or full path name.	**profdir.***task_id*

Table 58 summarizes the environment variables and flags for the Message Passing Interface. It includes information about how to set each variable, the values that may be specified, and the default value. These environment variables and flags allow you to change message and memory sizes, as well as other message passing information.

Table 58. POE environment variables and command line flags for Message Passing Interface (MPI)

Environment Variable Command Line Flag	Set:	Possible Values:	Default:
MP_ACK_THRESH -ack_thresh	Allows the user to control the packet acknowledgement threshold. Specify a positive integer.	A positive integer limited to 31.	**30**
MP_BUFFER_MEM -buffer_mem	See "MP_BUFFER_MEM details" on page 239.		**64 MB** (User Space and IP)
MP_CC_BUF_MEM -cc_buf_mem	See "MP_CC_BUF_MEM details" on page 240.		**4 MB** (User Space and IP)
MP_CC_SCRATCH_BUF -cc_scratch_buf	Use the fastest collective communication algorithm even if that algorithm requires allocation of more scratch buffer space.	**yes** **no**	**yes**

Table 58. POE environment variables and command line flags for Message Passing Interface (MPI) (continued)

Environment Variable Command Line Flag	**Set:**	**Possible Values:**	**Default:**
MP_CLOCK_SOURCE -clock_source	For AIX, to use the IBM High Performance Switch clock as a time source. For Linux, to specify the time source. Currently, the only valid value for Linux is **OS** (operating system).	For AIX: **AIX** **SWITCH** For Linux: **OS**	For AIX, None. See the table entitled *How the clock source is determined* in *IBM Parallel Environment: MPI Programming Guide* for more information. For Linux, **OS**
MP_CSS_INTERRUPT -css_interrupt	To specify whether or not arriving packets generate interrupts. Using this environment variable may provide better performance for certain applications. Setting this variable explicitly will suppress the MPI-directed switching of interrupt mode, leaving the user in control for the rest of the run. For more information, refer to the**MPI_FILE_OPEN** and **MPI_WIN_CREATE** subroutines in *IBM Parallel Environment: MPI Subroutine Reference*.	**yes** **no**	**no**

Table 58. POE environment variables and command line flags for Message Passing Interface (MPI) (continued)

Environment Variable Command Line Flag	Set:	Possible Values:	Default:
MP_EAGER_LIMIT -eager_limit	To change the threshold value for message size, above which rendezvous protocol is used. To ensure that at least 32 messages can be outstanding between any two tasks, **MP_EAGER_LIMIT** will be adjusted based on the number of tasks, when the user has specified neither **MP_BUFFER_MEM** nor **MP_EAGER_LIMIT**. For specific information about the default eager limit values, see *IBM Parallel Environment: MPI Programming Guide*. The maximum value for **MP_EAGER_LIMIT** is 256 KB (262144 bytes). Any value that is less than 64 bytes but greater than zero bytes is automatically increased to 64 bytes. A value of zero bytes is valid, and indicates that eager send mode is not to be used for the job. A non-power of 2 value will be rounded up to the nearest power of 2. A value may be adjusted if the early arrival buffer (**MP_BUFFER_MEM**) is too small.	*nnnnn* *nn*K (where: K = 1024 bytes)	**4096**
MP_HINTS_FILTERED -hints_filtered	To specify whether or not MPI **info** objects reject hints (*key* and *value* pairs) that are not meaningful to the MPI implementation.	**yes** **no**	**no**
MP_IONODEFILE -ionodefile	To specify the name of a parallel I/O node file — a text file that lists the nodes that should be handling parallel I/O. Setting this variable enables you to limit the number of nodes that participate in parallel I/O and guarantees that all I/O operations are performed on the same node. See "Determining which nodes will participate in parallel file I/O" on page 52 for more information.	Any relative path name or full path name.	None. All nodes will participate in parallel I/O.

Table 58. POE environment variables and command line flags for Message Passing Interface (MPI) (continued)

Environment Variable **Command Line Flag**	**Set:**	**Possible Values:**	**Default:**
MP_MSG_ENVELOPE_BUF -msg_envelope_buf	The size of the message envelope buffer (that is, uncompleted send and receive descriptors).	Any positive number. There is no upper limit, but any value less than 1 MB is ignored.	**8 MB**
MP_POLLING_INTERVAL -polling_interval	To change the polling interval (in microseconds).	An integer between 1 and 2 billion.	**400000**
MP_RETRANSMIT_INTERVAL -retransmit_interval	MP_RETRANSMIT_ INTERVAL=*nnnn* and its command line equivalent, -retransmit_interval=*nnnn*, control how often the communication subsystem library checks to see if it should retransmit packets that have not been acknowledged. The value *nnnn* is the number of polling loops between checks.	The acceptable range is from 1000 to INT_MAX	**10000** (for IP) **400000** (for User Space)
MP_LAPI_TRACE_LEVEL (no associated command line flag)	Used for debug purposes. For AIX, **MPI_LAPI_TRACE_LEVEL** is used in conjunction with AIX tracing. Levels 0-5 are supported.	**0** **1** **2** **3** **4** **5**	**0**

Table 58. POE environment variables and command line flags for Message Passing Interface (MPI) (continued)

Environment Variable **Command Line Flag**	**Set:**	**Possible Values:**	**Default:**
MP_USE_BULK_XFER -use_bulk_xfer	Exploits the IBM High Performance Switch and InfiniBand data transfer mechanisms. In other environments, this variable does not have any meaning and is ignored. Before you can use **MP_USE_BULK_XFER**, the system administrator must first enable Remote Direct Memory Access (RDMA). For more information, see *IBM Parallel Environment: Installation*. In other environments, this variable has no meaning and is ignored. When you use this environment variable, you also need to consider the value of the **MP_BULK_MIN_MSG_SIZE** environment variable. Messages with lengths that are greater than the value specified **MP_BULK_MIN_MSG_SIZE** will use the bulk transfer path, if it is available. For more information, see the entry for **MP_BULK_MIN_MSG_SIZE** in this table.	**yes** **no**	**no**
MP_BULK_MIN_MSG_SIZE -bulk_min_msg_size	Contiguous messages with data lengths greater than or equal to the value you specify for this environment variable will use the bulk transfer path, if it is available. Messages with data lengths that are smaller than the value you specify for this environment variable, or are noncontiguous, will use packet mode transfer.	The acceptable range is from 4096 to 2147483647 (INT_MAX). **Possible values:** *nnnnn (byte)* *nnnK* (where: K = 1024 bytes) *nnM* (where: M = 1024*1024 bytes) *nnG* (where: G = 1 billion bytes)	**153600**
MP_RC_MAX_QP No associated command line flag	Specifies the maximum number of RC QPs that can be created.	Any positive integer.	**2147483647**

Table 58. POE environment variables and command line flags for Message Passing Interface (MPI) (continued)

Environment Variable **Command Line Flag**	**Set:**	**Possible Values:**	**Default:**
MP_RC_USE_LMC No associated command line flag	Determines whether LMC (Lid Mask Control) is enabled. Enabling the use of LMC can improve performance, because a single port can support multiple RC paths. The default value is **no** (only one RC connected path is supported). Setting **MP_RC_USE_LMC** to **yes** causes multiple RC paths to be supported, which may improve performance.	**yes** **no**	**no**
MP_RDMA_COUNT -rdma_count **(Applies to PE for AIX only)**	To specify the number of user rCxt blocks. It supports the specification of multiple values when multiple protocols are involved.	*m* for a single protocol *m.n* for multiple protocols. The values are positional; *m* is for MPI, *n* is for LAPI. Only used when **MP_MSG_API=mpi.lapi**.	None
MP_SHARED_MEMORY -shared_memory	To specify the use of shared memory (instead of IP or the IBM High Performance Switch) for message passing between tasks running on the same node. **Note:** In past releases of PE for AIX, the **MP_SHM_CC** environment variable was used to enable or disable the use of shared memory for certain 64-bit MPI collective communication operations. Beginning with the PE 4.2 release, this environment variable has been removed. You should now use **MP_SHARED_MEMORY** to enable shared memory for both collective communication and point-to-point routines. The default setting for **MP_SHARED_MEMORY** is **yes** (enable shared memory).	**yes** **no**	**yes**

Table 58. POE environment variables and command line flags for Message Passing Interface (MPI) (continued)

Environment Variable Command Line Flag	**Set:**	**Possible Values:**	**Default:**
MP_SINGLE_THREAD -single_thread	To avoid lock overheads in a program that is known to be single-threaded. This is an optimization flag, with values of **no** and **yes**. The default value is **no**, which means the potential for multiple user message passing threads is assumed. Results are undefined if this variable is set to **yes** with multiple application message passing threads in use. To confirm that is it safe to run the application with **MP_SINGLE_THREAD** set to **yes**, run it once with **MP_SINGLE_THREAD** set to **confirm**. The **confirm** option can only warn you if this run has made MPI calls from more than a single thread. If an application is capable of both single-thread and multithread execution, **confirm** cannot warn you of the potential for multiple threads. If you want to use the PE barrier synchronization register (BSR), you must set **MP_SINGLE_THREAD** to **yes**. For more information about PE's support of the BSR, see *IBM Parallel Environment: MPI Programming Guide*.	**yes** **no** **confirm**	**no**
MP_THREAD_STACKSIZE -thread_stacksize	To specify the additional stack size allocated for user subroutines running on an MPI service thread. If you do not allocate enough space, the program may encounter a SIGSEGV exception or more subtle failures.	*nnnnn* *nnn*K (where: K = 1024 bytes) *nn*M (where: M = 1024*1024 bytes)	**0**

Table 58. POE environment variables and command line flags for Message Passing Interface (MPI) (continued)

Environment Variable Command Line Flag	Set:	Possible Values:	Default:
MP_UDP_PACKET_SIZE -udp_packet_size	Allows the user to control the packet size. Specify a positive integer.	A positive integer.	For AIX, **8K** if **MP_EUIDEVICE** is not set or is set to a value of **enX**. **64K** if **MP_EUIDEVICE** is set to value other than **enX**. For Linux, **1500**, if the value of **MP_EUIDEVICE** is prefixed by **eth**. Otherwise, the default value is **2044**.
MP_WAIT_MODE -wait_mode	**Set:** To specify how a thread or task behaves when it discovers it is blocked, waiting for a message to arrive.	**nopoll** **poll** **sleep** **yield**	**poll** (for User Space and IP)
MP_IO_BUFFER_SIZE -io_buffer_size	To specify the default size of the data buffer used by MPI-IO agents.	An integer less than or equal to 128 MB, in one of these formats: *nnnnn* *nnnK* (where K=1024 bytes) *nnnM* (where M=1024*1024 bytes)	The number of bytes that corresponds to 16 file blocks.
MP_IO_ERRLOG -io_errlog	To specify whether or not to turn on I/O error logging.	**yes** **no**	**no**
MP_REXMIT_BUF_SIZE -rexmit_buf_size	The maximum LAPI level message size that will be buffered locally, to more quickly free up the user send buffer. This sets the size of the local buffers that will be allocated to store such messages, and will impact memory usage, while potentially improving performance. The MPI application message size supported is smaller by, at most, 32 bytes.	*nnn* bytes (where: nnn > 0 bytes)	**16352** bytes
MP_REXMIT_BUF_CNT -rexmit_buf_cnt	The number of retransmit buffers that will be allocated per task. Each buffer is of size **MP_REXMIT_BUF_SIZE * MP_REXMIT_BUF_CNT**. This count controls the number of inflight messages that can be buffered to allow prompt return of application send buffers.	*nnn* (where: nnn > 0)	**128**

Table 58. POE environment variables and command line flags for Message Passing Interface (MPI) (continued)

Environment Variable Command Line Flag	**Set:**	**Possible Values:**	**Default:**
MP_SHMCC_EXCLUDE_LIST	To specify the collective communication routine in which the MPI level shared memory optimization should be disabled. The shared memory optimization in collective communication operations requires tight synchronization among parallel processes. Performance problems can arise if an application has an unbalanced workload such that processes participating in the same collective communication operation are out of synch. **MP_SHMCC_EXCLUDE_LIST** applies to 64-bit environments only.	**None** **All** Or a list of one or more of the following values, separated by a colon (:). **Barrier** **Bcast** **Reduce** **Allreduce** **Reduce_scatter** **Gather** **Gatherv** **Scatter** **Scatterv** **Allgather** **Allgatherv** **Alltoall** **Alltoallv** **Alltoallw** **Scan** **Exscan**	None

Table 59 summarizes the variables and flags for core file generation. It includes information about how to set each variable, the values that may be specified, and the default value.

Table 59. POE environment variables and command line flags for core file generation

The Environment Variable/Command Line Flag(s):	**Set:**	**Possible Values:**	**Default:**
MP_COREDIR -coredir	Creates a separate directory for each task's core file.	• Any valid directory name • **none** (to bypass creating a new directory)	*coredir.taskid*
MP_COREFILE_ FORMAT -corefile_format **(Applies to PE for AIX only)**	The format of core files generated when processes terminate abnormally.	**STDERR** (to specify that the lightweight core file information should be written to standard error) Any other string (to specify the lightweight core file name).	If not set/specified, standard AIX core files will be generated.
MP_COREFILE_ SIGTERM -corefile_sigterm **(Applies to PE for AIX only)**	Determines if POE should generate a core file when a **SIGTERM** signal is received. Valid values are **yes** and **no**. If not set, the default is **no**.	**yes** **no**	**no**

Table 60 summarizes some miscellaneous environment variables and flags. It includes information about how to set each variable, the values that may be specified, and the default value. These environment variables and flags enable additional error checking and let you set a dispatch priority class for execution.

Table 60. Other POE environment variables and command line flags

The Environment Variable/Command Line Flag(s):	**Set:**	**Possible Values:**	**Default:**
MP_COMPILER -compiler **(Applies to PE for Linux only)**	Specifies the name of the compiler to use. The default value is **xlc_r** (for C), **xlC_r** (for C++), **xlf_r** (for Fortran). If the IBM C **Set++ vac.cmp** package is not installed, the default value becomes one of the GNU compilers: **gcc** (for C), **g++** (for C++), or **g77** (for Fortran). You can use **MP_COMPILER** to specify a compiler other than the default. For example, you may want to use the GNU G++ compiler, even if the IBM C **Set++ vac.cmp** package is installed. Or, you may want to specify one of the other compilers that are contained in the IBM C **Set++ vac.cmp** package. For Fortran 90, specify the **xlf90** compiler.	Any of the C, C++, or Fortan compilers contained in the IBM C **Set++ vac.cmp** package. To specify a third-party compiler, you must either specify its full path or add that compiler to a directory in your search path (for example, **/usr/bin**).	For C; **xlc_r** | **gcc** For C++: **xlC_r** | **g++** For Fortran: **xlf_r** | **g77**
MP_EUIDEVELOP -euidevelop	Controls the level of parameter checking during execution. Setting this to yes enables some intertask parameter checking which may help uncover certain problems, but slows execution. Normal mode does only relatively inexpensive, local parameter checking. Setting this variable to *min* allows PE MPI to bypass parameter checking on all send and receive operations. **yes** or **deb** (debug) checking is intended for developing applications, and can significantly slow performance. **min** should only be used with well tested applications because a bug in an application running with **min** will not provide useful error feedback.	**yes** (develop mode) **no** or **nor** (normal mode) **deb** (debug mode) **min** (minimum mode)	**no**
MP_STATISTICS -statistics	Provides the ability to gather communication statistics for User Space jobs.	**yes** **no** **print**	**no**
MP_FENCE (no associated command line flag)	A **fence** character string for separating arguments you want parsed by POE from those you do not.	Any string.	None

Table 60. Other POE environment variables and command line flags (continued)

The Environment Variable/Command Line Flag(s):	**Set:**	**Possible Values:**	**Default:**
MP_NOARGLIST (no associated command line flag)	Whether or not POE ignores the argument list. If set to **yes**, POE will not attempt to remove POE command line flags before passing the argument list to the user's program.	**yes** **no**	**no**
MP_PRIORITY (no associated command line flag)	A dispatch priority class for execution or a string of high/low priority values. See *IBM Parallel Environment: Installation* for more information on dispatch priority classes.	Any of the dispatch priority classes set up by the system administrator or a string of high/low priority values in the file **/etc/poe.priority**. **Note:** If your cluster does not have a global time source (for example, an HPS switch), software synchronization of the node clocks (for example, NTP) is required. Otherwise, the high-priority and low-priority windows might not be sufficiently aligned, causing the coscheduler to be ineffective.	None
MP_PRIORITY_LOG -priority_log	Determines whether or not diagnostic messages should be logged to the POE priority adjustment coscheduler log file in **/tmp/log** on each of the remote nodes. This variable should only be used in conjunction with the POE coscheduler **MP_PRIORITY** variable. The value of this environment variable can be overridden using the **-priority_log** flag.	**yes** **no**	**no**

Table 60. Other POE environment variables and command line flags (continued)

The Environment Variable/Command Line Flag(s):	**Set:**	**Possible Values:**	**Default:**
MP_PRIORITY_LOG_DIR -priority_log_dir	Specifies the directory, on each of the remote nodes, into which the POE priority adjustment coscheduler log file is stored. The default directory is **/tmp**. The name of the log file is **pmadjpri.***jobid***.log**, by default, so after issuing this environment variable, the log file is stored in */directory_name/***pmadjpri.***jobid***.log**. If the specified directory is invalid, or cannot be written to, the default directory is used. To specify a name other than **pmadjpri.***jobid***.log**, use the **MP_PRIORITY_LOG_NAME** environment variable. This variable should only be used when the POE coscheduler **MP_PRIORITY_LOG** variable is set to **yes**.	Any full path name or relative path name. Must be a continuous ASCII string, containing no imbedded blanks.	**/tmp**
MP_PRIORITY_LOG_NAME -priority_log_name	Specifies the name of the POE priority adjustment coscheduler log file. The default name is **pmadjpri.***jobid***.log**. The directory into which this log file is placed is **/tmp**, by default, so after issuing this environment variable, the log file is stored in **/tmp/***log_file_name***.** *jobid***.log**. To specify a directory other than **/tmp**, use the **MP_PRIORITY_LOG_DIR** environment variable. This variable should only be used when the POE coscheduler **MP_PRIORITY_LOG** variable is set to **yes**.	Any file specifier, with a suffix of **.log**. Must be a continuous ASCII string, containing no imbedded blanks.	**pmadjpri.log**

Table 60. Other POE environment variables and command line flags (continued)

The Environment Variable/Command Line Flag(s):	Set:	Possible Values:	Default:
MP_PRIORITY_NTP -priority_ntp **(Applies to PE for AIX only)**	Determines whether the POE priority adjustment coscheduler will turn NTP off during the priority adjustment period, or leave it running. The value of **no** (which is the default) instructs the POE coscheduler to turn the NTP daemon off (if it was running) and restart NTP later, after the coscheduler completes. Specify a value of **yes** to inform the coscheduler to keep NTP running during the priority adjustment cycles (if NTP was not running, NTP will not be started). If **MP_PRIORITY_NTP** is not set, the default is **no**. The value of this environment variable can be overridden using the **-priority_ntp** flag.	**yes** **no**	**no**
MP_PRINTENV -printenv	Whether to produce a report of the current settings of MPI environment variables, across all tasks in a job. If **yes** is specified, the MPI environment variable information is gathered at initialization time from all tasks, and forwarded to task 0, where the report is prepared. If a *script_name* is specified, the script is run on each node, and the output script is forwarded to task 0 and included in the report. When a variable's value is the same for all tasks, it is printed only once. If it is different for some tasks, an asterisk (*) appears in the report after the word "Task".	**no** Do not produce a report of MPI environment variable settings. **yes** Produce a report of MPI environment variable settings. *script_name* Produce the report (same as **yes**), then run the script specified here.	**no**
MP_TLP_REQUIRED -tlp_required **(Applies to PE for AIX only)**	Specifies to POE whether to check to see if jobs being executed have been compiled for large pages, and when it finds a job that was not, the action to take.	**none** POE takes no action. **warn** POE detects and issues a warning message for any job that was not compiled for large pages. **kill** POE to detects and kills any job that was not compiled for large pages.	**none**

MP_BUFFER_MEM details

The **MP_BUFFER_MEM** environment variable allows you to control the amount of memory PE MPI allows for the buffering of early arrival message data for **point-to-point communications**. Message data that is sent without knowing if the receive is posted is said to be sent **eagerly**. If the message data arrives before the receive is posted, this is called an **early arrival** and must be buffered at the receive side.

Note: The **MP_CC_BUF_MEM** environment variable can be used to control the amount of memory that PE allows for the buffering of early arrival messages for **collective communications**. For more information, see "MP_CC_BUF_MEM details" on page 240.

Set:

There are two ways that the **MP_BUFFER_MEM** environment variable can be used:

1. To specify the pool size for memory to be allocated at MPI initialization time and dedicated to buffering of early arrivals. Management of pool memory for each early arrival is fast, which helps performance, but memory that is set aside in this pool is not available for other uses. Eager sending is throttled by PE MPI to be certain there will never be an early arrival that cannot fit **within the pool**. (To **throttle** a car engine is to choke off its air and fuel intake by lifting your foot from the gas pedal when you want to keep the car from going faster than you can control).
2. To specify the pool size for memory to be allocated at MPI initialization time and, with a second argument, an upper bound of memory to be used if the preallocated pool is not sufficient. Eager sending is throttled to be certain there will never be an early arrival that cannot fit **within the upper bound**. Any early arrival will be stored in the preallocated pool using its faster memory management if there is room, but if not, malloc and free will be used.

   The constraints on eager send must be pessimistic because they must guarantee an early arrival buffer no matter how the application behaves. Real applications at large task counts may suffer performance loss due to pessimistic throttling of eager sending, even though the application has only a modest need for early arrival buffering.

   Setting a higher bound allows more and larger messages to be sent eagerly. If the application is well behaved, it is likely that the preallocated pool will supply all the buffer space needed. If not, malloc and free will be used but never beyond the stated upper bound.

Possible values:

nnnnn (byte)
*nnn*K (where: K = 1024 bytes)
*nn*M (where: M = 1024*1024 bytes)
*nn*G (where: G = 1024*1024*1024 bytes)

Formats:

M1
M1,M2
,M2 (a comma followed by the **M2** value)

M1 specifies the size of the pool to be allocated at initialization time. **M1** must be between 0 and 256 MB.

M2 specifies the upper bound of memory that PE MPI will allow to be used for early arrival buffering in the most extreme case of sends without waiting receives. PE MPI will throttle senders back to rendezvous protocol (stop trying to use eager send) before allowing the early arrivals at a receive side to overflow the upper bound.

There is no limit enforced on the value you can specify for **M2**, but be aware that a program that does not behave as expected has the potential to malloc this much memory, and terminate if it is not available.

When **MP_BUFFER_MEM** is allowed to default, or is specified with a single argument, **M1**, the upper bound is set to the pool size, and eager sending will be throttled soon enough at each sender to ensure that the buffer pool cannot overflow at any receive side. If **M2** is smaller than **M1**, **M2** is ignored.

The format that omits **M1** is used to tell PE MPI to use its default size preallocated pool, but set the upper bound as specified with **M2**. This removes the need for a user to remember the default **M1** value when the intention is to only change the **M2** value.

It is expected that only jobs with hundreds of tasks will have any need to set **M2**. For most of these jobs, there will be an **M1,M2** setting that eliminates the need for PE MPI to throttle eager sends, while allowing all early arrivals that the application actually creates to be buffered within the preallocated pool.

IMPORTANT

The default size of the Early Arrival buffer has been changed from 2.8 MB to 64 MB for 32-bit IP applications. This is important to note, because the new default could cause your application to fail due to insufficient memory. As a result, you may need to adjust your application's memory allocation. For more information, see "PE Version 5 Release 1 migration information" on page 4.

MP_CC_BUF_MEM details

The **MP_CC_BUF_MEM** environment variable allows you to control the amount of memory PE MPI allows for the buffering of early arrival message data for **collective communications**. Message data that is sent without knowing if the receive is posted is said to be sent **eagerly**. If the message data arrives before the receive is posted, this is called an **early arrival** and must be buffered at the receive side.

Note: In PE 5.1, the early arrival buffer that is controlled by **MP_CC_BUF_MEM** is used by MPI_Bcast only. Early arrival messages in other collective communication operations continue to use the early arrival buffer for point-to-point communication that is controlled by **MP_BUFFER_MEM**.

Note: The **MP_BUFFER_MEM** environment variable can be used to control the amount of memory that PE allows for the buffering of early arrival messages for **point-to-point communications**. For more information, see "MP_BUFFER_MEM details" on page 239.

Set:

There are three ways that the **MP_CC_BUF_MEM** environment variable can be used:

1. To specify the pool size for memory to be allocated at MPI initialization time and dedicated to buffering of early arrivals. Management of pool memory for each early arrival is fast, which helps performance, but memory that is set aside in this pool is not available for other uses. Eager sending is throttled by PE MPI to be certain there will never be an early arrival that cannot fit **within the pool**. (To **throttle** a car engine is to choke off its air and fuel intake by lifting your foot from the gas pedal when you want to keep the car from going faster than you can control).
2. To specify an upper bound of memory to be used if the preallocated pool is not sufficient. Eager sending is throttled to be certain there will never be an early arrival that cannot fit **within the upper bound**.
3. To specify both the pool size for memory to be allocated at MPI initialization time and, with a second argument, an upper bound of memory to be used if the preallocated pool is not sufficient. Any early arrival will be stored in the preallocated pool using its faster memory management if there is room, but if not, malloc and free will be used.

   The constraints on eager send must be pessimistic because they must guarantee an early arrival buffer no matter how the application behaves. Real applications at large task counts may suffer performance loss due to pessimistic throttling of eager sending, even though the application has only a modest need for early arrival buffering.

   Setting a higher bound allows more messages to be sent eagerly. If the application is well behaved, it is likely that the preallocated pool will supply all the buffer space needed. If not, malloc and free will be used but never beyond the stated upper bound.

Possible values:

nnnnn (byte)
*nnn*K (where: K = 1024 bytes)
*nn*M (where: M = 1024*1024 bytes)
*nn*G (where: G = 1024*1024*1024 bytes)

Formats:

M1
M1,M2
,M2 (a comma followed by the **M2** value)

M1 specifies the size of the pool to be allocated at initialization time. **M1** must be between 4 and 128 MB.

M2 specifies the upper bound of memory that PE MPI will allow to be used for early arrival buffering in the most extreme case of sends without waiting receives (the default upper bound is 36 MB). PE MPI will throttle senders back to rendezvous protocol (stop trying to use eager send) before allowing the early arrivals at a receive side to overflow the upper bound.

There is no limit enforced on the value you can specify for **M2**, but be aware that a program that does not behave as expected has the potential to malloc this much memory, and terminate if it is not available.

When a value for only **M1** is specified, and the specified value is greater than 36 MB, the value of M2 is automatically increased to the same value.

The format that omits **M1** is used to tell PE MPI to use its default size preallocated pool, but set the upper bound as specified with **M2**. This removes the need for a user to remember the default **M1** value when the intention is to only change the **M2** value.

It is expected that only jobs that create a large number of communicators will have any need to set **M2**. For most of these jobs, there will be an **M1,M2** setting that eliminates the need for PE MPI to throttle eager sends, while allowing all early arrivals that the application actually creates to be buffered within the preallocated pool.

Appendix. Accessibility features for Parallel Environment

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in *Parallel Environment*:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers
- Keys that are discernible by touch but do not activate just by touching them
- Industry-standard devices for ports and connectors
- The attachment of alternative input and output devices

The *IBM Cluster information center*, and its related publications, are accessibility-enabled. The accessibility features of the information center are described at *http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.addinfo.doc/access.html*.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility:

`http://www.ibm.com/able`

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

All implemented function in the PE MPI product is designed to comply with the requirements of the Message Passing Interface Forum, MPI: A Message-Passing Interface Standard. The standard is documented in two volumes, Version 1.1, University of Tennessee, Knoxville, Tennessee, June 6, 1995 and *MPI-2: Extensions to the Message-Passing Interface*, University of Tennessee, Knoxville, Tennessee, July 18, 1997. The second volume includes a section identified as MPI 1.2 with clarifications and limited enhancements to MPI 1.1. It also contains the extensions identified as MPI 2.0. The three sections, MPI 1.1, MPI 1.2 and MPI 2.0 taken together constitute the current standard for MPI.

PE MPI provides support for all of MPI 1.1 and MPI 1.2. PE MPI also provides support for all of the MPI 2.0 Enhancements, except the contents of the chapter titled *Process Creation and Management*.

If you believe that PE MPI does not comply with the MPI standard for the portions that are implemented, please contact IBM Service.

Trademarks

Glossary

This glossary defines technical terms used in the IBM Parallel Environment documentation. If you do not find the term you are looking for, refer to the IBM Terminology site on the World Wide Web:

http://www.ibm.com/software/globalization/terminology/index.html

A

address. A unique code or identifier for a register, device, workstation, system, or storage location.

API. application programming interface (API): An interface that allows an application program that is written in a high-level language to use specific data or functions of the operating system or another program.

application. One or more computer programs or software components that provide a function in direct support of a specific business process or processes.

argument. A value passed to or returned from a function or procedure at run time.

authentication. The process of validating the identity of a user or server.

authorization. The process of obtaining permission to perform specific actions.

B

bandwidth. A measure of frequency range, typically measured in hertz. Bandwidth also is commonly used to refer to data transmission rates as measured in bits or bytes per second.

blocking operation. An operation that has not completed until the operation either succeeds or fails. For example, a blocking receive will not return until a message is received or until the channel is closed and no further messages can be received.

breakpoint. A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

broadcast. The simultaneous transmission of data to more than one destination.

C

C. A programming language designed by Bell Labs in 1972 for use as the systems language for the UNIX operating system.

C++. An enhancement of the C language that adds features supporting object-oriented programming.

client. A software program or computer that requests services from a server.

cluster. A group of processors interconnected through a high-speed network that can be used for high-performance computing.

collective communication. A communication operation that involves more than two processes or tasks. Broadcasts and reductions are examples of collective communication operations. All tasks in a communicator must participate.

communicator. A Message Passing Interface (MPI) object that describes the communication context and an associated group of processes.

compile. translate all or part of a program expressed in a high-level language into a computer program expressed in an intermediate language, an assembly language, or a machine language.

condition. One of a set of specified values that a data item can assume.

core dump. A process by which the current state of a program is preserved in a file. Core dumps are usually associated with programs that have encountered an unexpected, system-detected fault, such as a segmentation fault or a severe user error. A programmer can use the core dump to diagnose and correct the problem.

core file. A file that preserves the state of a program, usually just before a program is terminated because of an unexpected error. See also *core dump.*

D

data parallelism. A situation in which parallel tasks perform the same computation on different sets of data.

debugger. A tool used to detect and trace errors in computer programs.

distributed shell (dsh). A Cluster Systems Management (CSM) command that lets you issue commands to a group of hosts in parallel. See *IBM Cluster Systems Management: Command and Technical Reference* for details.

E

environment variable. (1) A variable that defines an aspect of the operating environment for a process. For example, environment variables can define the home directory, the command search path, the terminal in use, or the current time zone. (2) A variable that is included in the current software environment and is therefore available to any called program that requests it.

Ethernet. A packet-based networking technology for local area networks (LANs) that supports multiple access and handles contention by using Carrier Sense Multiple Access with Collision Detection (CSMA/CD) as the access method. Ethernet is standardized in the IEEE 802.3 specification.

executable program. A program that can be run as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms.

execution. The process of carrying out an instruction or instructions of a computer program by a computer.

F

fairness. A policy in which tasks, threads, or processes must eventually gain access to a resource for which they are competing. For example, if multiple threads are simultaneously seeking a lock, no set of circumstances can cause any thread to wait indefinitely for access to the lock.

Fiber Distributed Data Interface (FDDI). An American National Standards Institute (ANSI) standard for a 100-Mbps LAN using fiber optic cables.

file system. The collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

fileset. (1) An individually-installable option or update. Options provide specific function, and updates correct an error in, or enhance, a previously installed program. (2) One or more separately-installable, logically-grouped units in an installation package. See also *licensed program* and *package.*

FORTRAN. A high-level programming language used primarily for scientific, engineering, and mathematical applications.

G

GDB. An open-source portable debugger supporting Ada, C, C++, and FORTRAN. GDB is a useful tool for determining why a program crashes and where, in the program, the problem occurs.

global max. The maximum value across all processors for a given variable. It is global in the sense that it is global to the available processors.

global variable. A symbol defined in one program module that is used in other program modules that are independently compiled.

graphical user interface (GUI). A type of computer interface that presents a visual metaphor of a real-world scene, often of a desktop, by combining high-resolution graphics, pointing devices, menu bars and other menus, overlapping windows, icons and the object-action relationship.

GUI. See graphical user interface.

H

high performance switch. A high-performance message-passing network that connects all processor nodes.

home node. The node from which an application developer compiles and runs a program. The home node can be any workstation on the LAN.

host. A computer that is connected to a network and provides an access point to that network. The host can be a client, a server, or both a client and server simultaneously.

host list file. A file that contains a list of host names, and possibly other information. The host list file is defined by the application that reads it.

host name. The name used to uniquely identify any computer on a network.

I

installation image. A copy of the software, in backup format, that the user is installing, as well as copies of other files the system needs to install the software product.

Internet. The collection of worldwide networks and gateways that function as a single, cooperative virtual network.

Internet Protocol (IP). A protocol that routes data through a network or interconnected networks. This protocol acts as an intermediary between the higher protocol layers and the physical network.

IP. Internet Protocol.

K

kernel. The part of an operating system that contains programs for such tasks as input/output, management and control of hardware, and the scheduling of user tasks.

L

latency. The time from the initiation of an operation until something actually starts happening (for example, data transmission begins).

licensed program. A separately priced program and its associated materials that bear a copyright and are offered to customers under the terms and conditions of a licensing agreement.

lightweight core files. An alternative to standard AIX core files. Core files produced in the Standardized Lightweight Corefile Format provide simple process stack traces (listings of function calls that led to the error) and consume fewer system resources than traditional core files.

LoadLeveler pool. A group of resources with similar characteristics and attributes.

local variable. A symbol defined in one program module or procedure that can only be used within that program module or procedure.

M

management domain . A set of nodes that are configured for management by Cluster Systems Management. Such a domain has a management server that is used to administer a number of managed nodes. Only management servers have knowledge of the domain. Managed nodes only know about the servers managing them.

menu. A displayed list of items from which a user can make a selection.

message catalog. An indexed table of messages. Two or more catalogs can contain the same index values. The index value in each table refers to a different language version of the same message.

message passing. The process by which parallel tasks explicitly exchange program data.

Message Passing Interface (MPI). A library specification for message passing. MPI is a standard application programming interface (API) that can be used by parallel applications.

MIMD. multiple instruction stream, multiple data stream.

multiple instruction stream, multiple data stream (MIMD). A parallel programming model in which different processors perform different instructions on different sets of data.

MPMD. Multiple program, multiple data.

Multiple program, multiple data (MPMD). A parallel programming model in which different, but related, programs are run on different sets of data.

N

network. In data communication, a configuration in which two or more locations are physically connected for the purpose of exchanging data.

network information services (NIS). A set of network services (for example, a distributed service for retrieving information about the users, groups, network addresses, and gateways in a network) that resolve naming and addressing differences among computers in a network.

NIS. See *network information services*.

node ID. A string of unique characters that identifies the node on a network.

nonblocking operation. An operation, such as sending or receiving a message, that returns immediately whether or not the operation has completed. For example, a nonblocking receive does not wait until a message arrives. A nonblocking receive must be completed by a later test or wait.

O

object code. Machine-executable instructions, usually generated by a compiler from source code written in a higher level language. Object code might itself be executable or it might require linking with other object code files.

optimization. The process of achieving improved run-time performance or reduced code size of an application. Optimization can be performed by a compiler, by a preprocessor, or through hand tuning of source code.

option flag. Arguments or any other additional information that a user specifies with a program name. Also referred to as parameters or command line options.

P

package. 1) In AIX, a number of filesets that have been collected into a single installable image of licensed programs. See also fileset and licensed program. 2) In

Linux, a collection of files, usually used to install a piece of software. The equivalent AIX term is fileset.

parallelism. The degree to which parts of a program may be concurrently executed.

parallelize. To convert a serial program for parallel execution.

parameter. A value or reference passed to a function, command, or program that serves as input or controls actions. The value is supplied by a user or by another program or process.

peer domain. A set of nodes configured for high availability. Such a domain has no distinguished or master node. All nodes are aware of all other nodes, and administrative commands can be issued from any node in the domain. All nodes also have a consistent view of the domain membership. Contrast with *management domain*.

point-to-point communication. A communication operation that involves exactly two processes or tasks. One process initiates the communication through a send operation. The partner process issues a receive operation to accept the data being sent.

procedure. In a programming language, a block, with or without formal parameters, that is initiated by means of a procedure call. (2) A set of related control statements that cause one or more programs to be performed.

process. A program or command that is actually running the computer. A process consists of a loaded version of the executable file, its data, its stack, and its kernel data structures that represent the process's state within a multitasking environment. The executable file contains the machine instructions (and any calls to shared objects) that will be executed by the hardware. A process can contain multiple threads of execution.

The process is created with a **fork()** system call and ends using an **exit()** system call. Between **fork** and **exit**, the process is known to the system by a unique process identifier (PID).

Each process has its own virtual memory space and cannot access another process's memory directly. Communication methods across processes include pipes, sockets, shared memory, and message passing.

profiling. A performance analysis process that is based on statistics for the resources that are used by a program or application.

pthread. A shortened name for the i5/OS threads API set that is based on a subset of the POSIX standard.

R

reduction operation. An operation, usually mathematical, that reduces a collection of data by one or more dimensions. For example, an operation that reduces an array to a scalar value.

remote host. Any host on a network except the host at which a particular operator is working.

remote shell (rsh). A variant of the remote login (rlogin) command that invokes a command interpreter on a remote UNIX machine and passes the command-line arguments to the command interpreter, omitting the login step completely.

RSCT peer domain. See *peer domain*.

S

secure shell (ssh). A Unix-based command interface and protocol for securely accessing a remote computer.

shell script. A program, or script, that is interpreted by the shell of an operating system.

segmentation fault. A system-detected error, usually caused by a reference to a memory address that is not valid.

server. A software program or a computer that provides services to other software programs or other computers.

single program, multiple data (SPMD). A parallel programming model in which different processors run the same program on different sets of data.

source code. A computer program in a format that is readable by people. Source code is converted into binary code that can be used by a computer.

source line. A line of source code.

SPMD. single program, multiple data.

standard error (STDERR). The output stream to which error messages or diagnostic messages are sent.

standard input (STDIN). An input stream from which data is retrieved. Standard input is normally associated with the keyboard, but if redirection or piping is used, the standard input can be a file or the output from a command.

standard output (STDOUT). The output stream to which data is directed. Standard output is normally associated with the console, but if redirection or piping is used, the standard output can be a file or the input to a command.

STDERR. standard error.

STDIN. standard input.

STDOUT. standard output.

subroutine. A sequence of instructions within a larger program that performs a particular task. A subroutine can be accessed repeatedly, can be used in more than one program, and can be called at more than one point in a program.

synchronization. The action of forcing certain points in the execution sequences of two or more asynchronous procedures to coincide in time.

system administrator. The person who controls and manages a computer system.

T

task. In a parallel job, there are two or more concurrent tasks working together through message passing. Though it is common to allocate one task per processor, the terms *task* and *processor* are not interchangeable.

thread. A stream of computer instructions. In some operating systems, a thread is the smallest unit of operation in a process. Several threads can run concurrently, performing different jobs.

trace. A record of the processing of a computer program or transaction. The information collected from a trace can be used to assess problems and performance.

U

user. (1) An individual who uses license-enabled software products. (2) Any individual, organization, process, device, program, protocol, or system that uses the services of a computing system.

User Space. A version of the message passing library that is optimized for direct access to the high performance switch (PE for AIX) or communication adapter (PE for Linux). User Space maximizes performance by not involving the kernel in sending or receiving a message.

utility program. A computer program in general support of computer processes; for example, a diagnostic program, a trace program, a sort program.

utility routine. A routine in general support of the processes of a computer; for example, an input routine.

V

variable. A representation of a changeable value.

X

X Window System. A software system, developed by the Massachusetts Institute of Technology, that enables the user of a display to concurrently use multiple application programs through different windows of the display. The application programs can execute on different computers.

Index

Reader’s Comments– We’d like to hear from you

Parallel Environment for AIX and Linux
Operation and Use
Version 5 Release 1

Publication No. SC23-6667-00

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send your comments via e-mail to: mhvrcfs@us.ibm.com

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	E-mail address

Readers' Comments — We'd Like to Hear from You
SC23-6667-00

Cut or Fold Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 58HA, Mail Station P181
2455 South Road
Poughkeepsie NY
12601-5400

Fold and Tape | **Please do not staple** | Fold and Tape

SC23-6667-00

Cut or Fold Along Line

Program Number: 5765-PEA and 5765-PEL

SC23-6667-00