IBM Parallel Environment for AIX 5L

Introduction

Version 4 Release 3.0

SA22-7947-05

IBM Parallel Environment for AIX 5L

Introduction

Version 4 Release 3.0

SA22-7947-05

Note

Before using this information and the product it supports, read the information in "Notices" on page 105.

Sixth Edition (October 2006)

This edition applies to version 4, release 3, modification 0 of IBM Parallel Environment for AIX 5L (product number 5765-F83) and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces SA22-7947-04. Significant changes or additions to the text and illustrations are indicated by a vertical line (|) to the left of the change.

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

International Business Machines Corporation
Department 55JA, Mail Station P384
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 would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:
Title and order number of this book
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

Figures

Tables

About this book

This book provides suggestions and guidance for using the IBM® Parallel Environment for AIX 5L™ (5765-F83) to develop and run Fortran, C, and C++ parallel applications. To make this book a little easier to read, the name *IBM Parallel Environment* has been abbreviated to *PE* throughout.

In this book, you will find information on basic parallel programming concepts and the Message Passing Interface (MPI) standard. You will also find information about the application development tools that are provided by PE such as the Parallel Operating Environment and the Parallel Debugger.

This book contains examples and illustrates various commands and programs as well as the output you receive as a result of running them. When looking at these examples, keep in mind that the output you see on the system may not exactly match what is printed in the book. The included examples give you a basic idea of what happens.

This book assumes that AIX 5L Version 5.3 Technology Level 5300-05 or higher, and the X-Windows system are already installed, if required.

Who should read this book

This book is intended for application developers who are interested in creating and running parallel programs. To make the best use of this book, you should be familiar with the following:
- The AIX® operating system
- One or more of the supported programming languages (Fortran, C, or C++)
- Basic parallel programming concepts.

This book is not intended to provide comprehensive coverage of the topics, nor is it intended to tell you everything there is to know about IBM Parallel Environment (PE). If you are new to either message passing parallel programming or to PE, you should find this book useful. For the latest information, always use the documents at:

`http://publib.boulder.ibm.com/infocenter/clresctr/index.jsp`

The purpose of this book is to get you started creating parallel programs with PE. Once you have mastered these initial concepts, you will need to know more about how PE works. For information on the Parallel Operating Environment (POE), see *IBM Parallel Environment: Operation and Use, Volume 1*. For information on PE tools, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*.

How this book is organized

Overview of contents

This book contains the following information:

- **Chapter 1, "Understanding the environment," on page 1** familiarizes you with the Parallel Operating Environment (POE).
- **Chapter 2, "Message passing," on page 21** covers parallelization techniques and discusses their advantages and disadvantages. It discusses how you take a working serial program and create a parallel program that gives the same result.

- **Chapter 3, "Diagnosing and correcting common problems," on page 35** outlines the possible causes for a parallel application to fail to execute correctly, and how the tools available with PE can be used to identify and correct problems.
- **Chapter 4, "Is the program efficient?," on page 61** discusses some of the ways you can optimize the performance of a parallel program and some hints on tuning the performance of the program.
- **Chapter 5, "Creating a safe program," on page 93** provides you with some general guidelines for creating *safe* parallel MPI programs.
- **Appendix A, "A sample program to illustrate messages," on page 97** provides a sample program, run with the maximum level of error messages. It points out the various types of messages you can expect, and tells you what they mean.
- **Appendix B, "Parallel Environment internals," on page 101** provides some additional information about how the PE works with respect to your application.

Conventions and terminology used in this book

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

This book 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.
\	The continuation character is used in coding examples in this book for formatting purposes.

In addition to the highlighting conventions, this manual 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 manual presents the instruction as:

ENTER
　　tool

Abbreviated names

Some of the abbreviated names used in this book follow.

AIX	Advanced Interactive Executive
CSM	Clusters Systems Management
CSS	communication subsystem
CTSEC	cluster-based security
DPCL	dynamic probe class library
dsh	distributed shell
GUI	graphical user interface
HDF	Hierarchical Data Format
IP	Internet Protocol
LAPI	Low-level Application Programming Interface
MPI	Message Passing Interface
NetCDF	Network Common Data Format
PCT	Performance Collection Tool
PE	IBM® Parallel Environment for AIX®
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
pSeries®	IBM eServer™ pSeries
PVT	Profile Visualization Tool
RISC	reduced instruction set computer
RSCT	Reliable Scalable Cluster Technology
rsh	remote shell
STDERR	standard error
STDIN	standard input
STDOUT	standard output
UTE	Unified Trace Environment
System x	IBM System x

Prerequisite and related information

The Parallel Environment for AIX library consists of:

- IBM Parallel Environment: Introduction, SA22-7947
- IBM Parallel Environment: Installation, GA22-7943
- IBM Parallel Environment: Operation and Use, Volume 1, SA22-7948
- IBM Parallel Environment: Operation and Use, Volume 2, SA22-7949
- IBM Parallel Environment: MPI Programming Guide, SA22-7945
- IBM Parallel Environment: MPI Subroutine Reference, SA22-7946
- IBM Parallel Environment: Messages, GA22-7944

To access the most recent Parallel Environment documentation in PDF and HTML format, refer to the IBM eServer Cluster Information Center on the Web at:

http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp

Both the current Parallel Environment books and earlier versions of the library are also available in PDF format from the IBM Publications Center Web site located at:

http://www.ibm.com/shop/publications/order/

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.

Using LookAt to look up message explanations

LookAt is an online facility that lets you look up explanations for most of the IBM messages you encounter, as well as for some system abends and codes. You can use LookAt from the following locations to find IBM message explanations for Clusters for AIX:

- The Internet. You can access IBM message explanations directly from the LookAt Web site:

  http://www.ibm.com/eserver/zseries/zos/bkserv/lookat/
- Your wireless handheld device. You can use the LookAt Mobile Edition with a handheld device that has wireless access and an Internet browser (for example, Internet Explorer for Pocket PCs, Blazer, or Eudora for Palm OS, or Opera for Linux® handheld devices). Link to the LookAt Mobile Edition from the LookAt Web site.

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 book 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 AIX 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:

ENTER

```
export NLSPATH=/usr/lib/nls/msg/%L/%N

export LANG=C
```

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

/usr/lib/nls/msg/C

/usr/lib/nls/msg/En_US

/usr/lib/nls/msg/en_US

If your site is using its own translations of the message catalogs, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For more information on NLS and message catalogs, see *AIX: General Programming Concepts: Writing and Debugging Programs*.

Summary of changes for Parallel Environment 4.3

This release of IBM Parallel Environment for AIX contains a number of functional enhancements, including:

- PE 4.3 supports only AIX 5L Version 5.3 Technology Level 5300-05, or later versions.

  AIX 5L Version 5.3 Technology Level 5300-05 is referred to as AIX 5L V5.3 TL 5300-05 or AIX 5.3.
- Support for Parallel Systems Support Programs for AIX (PSSP), the SP™ Switch2, POWER3™ servers, DCE, and DFS™ has been removed. PE 4.2 is the **last** release that supported these products.
- PE Benchmarker support for IBM System p5™ model 575 has been added.
- A new environment variable, **MP_TLP_REQUIRED** is available to detect the situation where a parallel job that should be using large memory pages is attempting to run with small pages.
- A new command, **rset_query**, for verifying that memory affinity assignments have been performed.
- Performance of MPI one-sided communication has been substantially improved.
- Performance improvements to some MPI collective communication subroutines.
- The default value for the **MP_BUFFER_MEM** environment variable, which specifies the size of the Early Arrival (EA) buffer, is now 64 MB for both IP and User Space. In some cases, 32 bit IP applications may need to be recompiled with more heap or run with **MP_BUFFER_MEM** of less than 64 MB. For more details, see the migration information in Chapter 1 of *IBM Parallel Environment: Operation and Use, Volume 1* and Appendix E of *IBM Parallel Environment: MPI Programming Guide*.

Chapter 1. Understanding the environment

To understand the new environment, IBM Parallel Environment for AIX (PE), you must understand these concepts:

- PE
- The Parallel Operating Environment (POE)
- Starting the POE
- Running simple commands
- Experimenting with parameters and environment variables
- Using a *host list* file versus a job management system (LoadLeveler) for requesting processor nodes
- Compiling and running a simple parallel application
- Some simple environment setup and debugging tips.

What is IBM Parallel Environment for AIX?

IBM Parallel Environment for AIX (PE) software lets you develop, debug, analyze, tune, and execute parallel applications written in Fortran, C, and C++. PE conforms to the MPI standard, except that PE does not include the functions defined by the chapter 'Process Creation and Management' of MPI-2 . PE commands and interfaces follow the POSIX model.

PE consists of the following:

- The Parallel Operating Environment (POE), for submitting and managing jobs.
- A message passing library (MPI), for communication among the tasks that make up a parallel program.
- A parallel debugger (pdbx for AIX, PDB for Linux) for debugging parallel programs.
- Parallel utilities for easing file manipulation.
- PE Benchmarker, a suite of applications and utilities you can use to analyze program performance.

What is the Parallel Operating Environment?

The Parallel Operating Environment (POE) allows you to develop and execute the parallel applications across multiple operating system images, called **nodes**. When using POE, there is a single node (possibly a workstation) that is called the *home node* that manages interactions with users.

POE transparently manages the allocation of remote nodes where the parallel application actually runs. It also handles the various requests and communication between the home node and the remote nodes via the underlying network.

This approach eases the transition from serial to parallel programming by hiding the differences, and allowing you to continue using standard AIX tools and techniques. You have to tell POE what remote nodes to use, but once you have, POE does the rest.

The *processor node* is a physical entity or operating system image that is defined to the network. It can be a standalone machine, or a processor node within a frame or clustered server, or an SMP node. From POE's point of view, a node is a single copy of the operating system.

Before you start

Before starting, check that you have addressed the following items:

Installation

The person who installed POE should have verified that it was installed successfully by running the *Installation Verification Program* (IVP). The IVP is discussed in *IBM Parallel Environment: Installation*.

The IVP tests to see if POE can do the following:
- Establish a remote execution environment
- Compile and execute the program
- Initialize the IP message passing environment
- Check that the MPI library is operable.

Access

Before running the job, you must first have access to computer resources in the system. Here are some things to consider:

- You must have the *same* user ID and group ID on the home node and each remote node on which you will be running the parallel application.
- POE will not allow you to run the application as root.

If you are using LoadLeveler to submit POE jobs, which includes all user space applications, then LoadLeveler is responsible for the security authentication. The security function in POE is not invoked when POE is run under LoadLeveler.

Security methods: PE uses an enhanced set of security methods, based on Cluster Security Services in RSCT (Reliable Scalable Cluster Technology). RSCT is a set of software components that provide a comprehensive clustering environment. RSCT is the infrastructure used by a variety of products to provide clusters with improved system availability, scalability, and ease of use. POE now has a security configuration option for the system administrator to determine which set of security methods are used in the system. Two types of security methods are supported:

- cluster security services
- AIX based security (or Compatibility, which is the default)

For more information about these security methods, and how to configure them, see the *IBM Parallel Environment: Installation*.

Cluster security services: When cluster based security is the security method of choice, the system administrator will have to ensure that UNIX® Host Based authentication is enabled and properly configured on all nodes. Refer to *IBM Parallel Environment: Installation* and *IBM Reliable Scalable Cluster Technology: Guide and Reference* for details.

When using cluster based security, users will be required to have the proper entries in the **/etc/hosts.equiv** or **.rhosts** files, to ensure proper access to each node, as described in "User authorization."

AIX based security: When AIX based security (compatibility) is the security method of choice (which is also the default), POE will rely on the authority as defined in "User authorization."

User authorization

You must have remote execution authority on all the nodes in the system that you will use for parallel execution. The system administrator should:

- Authorize both the home node machine and the user name (or machine names) in the **/etc/hosts.equiv** file on each remote node, or
- Set up the **.rhosts** file in the home directory of the user ID for each node that you want to use. The contents of each **.rhosts** file can be either the explicit IP address of the home node, or the home node name. For more information about **.rhosts** files, see the *IBM Parallel Environment: Installation*.

/etc/hosts.equiv is checked first, and, if the home node and user/machine name do not appear there, it then looks to **.rhosts**.

You can verify that you have remote execution authority by running a remote shell from the workstation where you intend to submit parallel jobs. For example, to test whether you have remote execution authority on node **202r1n10**, try the following command:

```
$ rsh 202r1n10 hostname
```

The response should be the remote host name. If it is not the remote host name, or the command cannot run, see the system administrator. Issue this command for every remote host on which you plan to have POE execute the job.

| POE does not use rsh, but does use the same user authentication mechanism. If
| rsh is not installed on your workstation, then you will not be able to run this test.

Refer to *IBM Parallel Environment: Installation* for more information on enabling **rsh**.

Host list file

One way to tell POE where to run the program is by using a *host list* file. The host list file is generally in the current working directory, but you can move it anywhere you like by specifying certain parameters. This file can be given any name, but the default name is *host.list*. Many people use *host.list* as the name to avoid having to specify another parameter. This file contains one of two different kinds of information; node names or pool numbers (a pool can also be designated by a string).

Node names refer to the hosts on which parallel jobs may be run. They may be specified as Domain Names (as long as those Domain Names can be resolved from the workstation where you submit the job) or as Internet addresses. Each host goes on a separate line in the host list file.

Here is an example of a host list file that specifies the node names on which four tasks will run:

```
202r1n09.hpssl.kgn.ibm.com
202r1n10.hpssl.kgn.ibm.com
202r1n11.hpssl.kgn.ibm.com
202r1n12.hpssl.kgn.ibm.com
```

Running POE

After you have checked all the items in “Before you start” on page 2, you are ready to run the POE. You can view POE as a way to run commands and programs on multiple nodes from a single point. Remember that these commands and programs are really running on the remote nodes. If you ask POE to perform some operation on a remote node, everything necessary to perform that operation must be available on the remote node.

There are two ways to influence the way the parallel program is executed; with environment variables or command line option flags. You can set environment variables at the beginning of the session to influence each program that you execute. You also get the same effect by specifying the related command line flag when you invoke POE, but its influence lasts only for that particular program execution. “Running POE with environment variables” on page 6 gives you some high-level information, but you may also want to refer to *IBM Parallel Environment: Operation and Use, Volume 1* to learn more about using environment variables.

Some examples of running POE

The **poe** command enables you to load and execute programs on remote nodes. The syntax is:

```
poe [program] [options]
```

When you invoke **poe**, it allocates processor nodes for each task and initializes the local environment. It then loads the program and reproduces the local shell environment on each processor node. POE also passes the user program arguments to each remote node.

The simplest thing to do with POE is to run a command. When you try these examples on the system, use a host list file that contains the node names (as opposed to a pool number). These examples assume at least a four-node parallel environment. If you have more than four nodes, feel free to use more. If you have fewer than four nodes, you may duplicate lines. This example assumes that the file is called *host.list*, and is in the directory from which you are submitting the parallel job. If either of these conditions are not true, POE will not find the host list file unless you use the **-hostfile** option.

The **-procs 4** option tells POE to run this command on four nodes. It will use the first four in the host list file.

```
$ poe hostname -procs 4

202r1n10.hpssl.kgn.ibm.com
202r1n11.hpssl.kgn.ibm.com
202r1n09.hpssl.kgn.ibm.com
202r1n12.hpssl.kgn.ibm.com
```

What you see is the output from the **hostname** command run on each of the remote nodes. POE has taken care of submitting the command to each node, collecting the standard output and standard error from each remote node, and sending it back to the workstation. One thing that you do not see is an indication of which task is responsible for each line of output. In a simple example like this, it is not that important. If, however, you had many lines of output from each node, you would want to know which task was responsible for each line of output. To do that, you use the **-labelio** option:

```
$ poe hostname -procs 4 -labelio yes

1:202r1n10.hpssl.kgn.ibm.com
2:202r1n11.hpssl.kgn.ibm.com
0:202r1n09.hpssl.kgn.ibm.com
3:202r1n12.hpssl.kgn.ibm.com
```

Notice how each line starts with a number and a colon and that the numbering started at 0 (zero). The number is the task ID that the line of output came from (it is also the line number in the host list file that identifies the host which generated this output). Use this parameter to identify lines from a command that generates more output. Try this command:

```
$ poe cat /etc/motd -procs 2 -labelio yes
```

You should see something similar to this:

```
0:*******************************************************************************
0:*                                                                             *
0:*  Welcome to IBM AIX Version 5.3   on 202r1n09.hpssl.kgn.ibm.com             *
0:*                                                                             *
0:*******************************************************************************
0:*                                                                             *
0:*     Message of the Day:  Never drink more than 3                            *
0:*     Blasters unless you are a 50 ton elephant.                              *
0:*                                                                             *
0:*                                                                             *
1:*******************************************************************************
1:*                                                                             *
1:*  Welcome to IBM AIX Version 5.3   on 202r1n10.hpssl.kgn.ibm.com             *
1:*                                                                             *
1:*******************************************************************************
1:*                                                                             *
1:*                                                                             *
1:*     Message of the Day:  Never drink more than 3                            *
1:*     Blasters unless you are a 50 ton elephant.                              *
1:*                                                                             *
1:*                                                                             *
1:*                                                                             *
1:*******************************************************************************
0:*                                                                             *
0:*                                                                             *
0:*                                                                             *
0:*******************************************************************************
```

The **cat** command is listing the contents of the file **/etc/motd** on each of the remote nodes. Notice how the output from each of the remote nodes is intermingled. This is because as soon as a buffer is full on the remote node, POE sends it back to the workstation for display (in case you had any doubts that these commands were really being executed in parallel). The result is the jumbled mess that can be difficult to interpret. Fortunately, POE can clear things up with the **-stdoutmode** parameter.

Try this command:

```
$ poe cat /etc/motd -procs 2 -labelio yes -stdoutmode ordered
```

You should see something similar to this:

```
0:*******************************************************************************
0:*                                                                             *
0:*  Welcome to IBM AIX Version 5.3   on 202r1n09.hpssl.kgn.ibm.com             *
0:*                                                                             *
0:*******************************************************************************
0:*                                                                             *
0:*                                                                             *
0:*     Message of the Day:  Never drink more than 3 Blasters                   *
0:*     unless you are a 50 ton elephant.                                       *
0:*                                                                             *
0:*                                                                             *
0:*                                                                             *
0:*******************************************************************************
1:*******************************************************************************
1:*                                                                             *
1:*  Welcome to IBM AIX Version 5.3   on 202r1n10.hpssl.kgn.ibm.com             *
1:*                                                                             *
1:*******************************************************************************
1:*                                                                             *
1:*                                                                             *
1:*     Message of the Day:  Never drink more than 3 Blasters                   *
```

```
1:*     unless you are a 50 ton elephant.                                      *
1:*                                                                            *
1:*                                                                            *
1:*                                                                            *
1:******************************************************************************
```

POE holds all the output until the jobs either finish or POE itself runs out of space. If the jobs finish, POE displays the output from each remote node together. If POE runs out of space, it prints everything, and then starts a new page of output. You get less of a sense of the parallel nature of the program, but it is easier to understand.

Running POE with environment variables

If you are getting tired of typing the same command line options over and over again, you can set them as environment variables so that you do not have to put them on the command line. The environment variable names are the same as the command line option names (without the leading dash), but they start with **MP_**, all in upper case. For example, the environment variable name for the **-procs** option is **MP_PROCS**, and for the **-labelio** option it is **MP_LABELIO**. Setting these two variables like this:

```
$ export MP_PROCS=2
$ export MP_LABELIO=yes
```

allows you to run the **/etc/motd** program with two processes and labeled output, without specifying either with the **poe** command.

Try this command:

```
$ poe cat /etc/motd -stdoutmode ordered
```

You should see something similar to this:

```
0:******************************************************************************
0:*                                                                            *
0:*  Welcome to IBM AIX Version 5.3   on pe03.pok.ibm.com                      *
0:*                                                                            *
0:******************************************************************************
0:*                                                                            *
0:*                                                                            *
0:*     Message of the Day:  Never drink more than 3 Blastes                   *
0:*     unless you are a 50 ton elephant.                                      *
0:*                                                                            *
0:*                                                                            *
0:*                                                                            *
0:******************************************************************************
1:******************************************************************************
1:*                                                                            *
1:*  Welcome to IBM AIX Version 5.3  on pe03.pok.ibm.com                       *
1:*                                                                            *
1:******************************************************************************
1:*                                                                            *
1:*                                                                            *
1:*     Message of the Day:  Never drink more than 3                           *
1:*     Blasters unless you are a 50 ton elephant.                             *
1:*                                                                            *
1:*                                                                            *
1:*                                                                            *
1:******************************************************************************
```

In the previous example, the program ran with two processes, and the output was labeled.

Now, to see that the environment variable setting lasts for the duration of the session, try running the command below, without specifying the number of processes or labeled I/O.

```
$ poe hostname

0:202r1n09.hpssl.kgn.ibm.com
1:202r1n10.hpssl.kgn.ibm.com
```

Notice that the program still ran with two processes and you got labeled output.

Now try overriding the environment variables just set. To do this, use command line options when running POE. Try running the following command:

```
$ poe hostname -procs 4 -labelio no

202r1n09.hpssl.kgn.ibm.com
202r1n12.hpssl.kgn.ibm.com
202r1n11.hpssl.kgn.ibm.com
202r1n10.hpssl.kgn.ibm.com
```

This time, notice that the program ran with four processes and that the output was not labeled. No matter what the environment variables have been set to, you can always override them when you run POE.

To show that this was a temporary override of the environment variable settings, try running the following command again, without specifying any command line options.

```
$ poe hostname

0:202r1n09.hpssl.kgn.ibm.com
1:202r1n10.hpssl.kgn.ibm.com
```

Once again, the program ran with two processes, and the output was labeled.

Compiling

You probably have programs that you want to run in parallel. Chapter 2, “Message passing,” on page 21 talks about creating parallel programs in a more detail. Right now the topic is compiling a program for POE. You can compile almost any Fortran, C, or C++ program for execution under POE.

Before compiling, you should verify that the following has happened:

- POE is installed on the system
- You are authorized to use POE
- A Fortran, C Compiler, or C ++ compiler is installed on the system.

See *IBM Parallel Environment: MPI Programming Guide* for information on compilation restrictions for POE.

This example, showing how compiling works, uses the *Hello World* program. Here it is in C:

```
/*********************************************************************
*
* Hello World C Example
*
* To compile:
* mpcc_r -o hello_world_c hello_world.c
*
**********************************************************************/
#include<stdlib.h>
#include<stdio.h>
```

```
/* Basic program to demonstrate compilation and execution techniques */
int main()
{
printf("Hello, World!\n");
exit(0);
}
```

And here it is in Fortran:

```
c*********************************************************************
c*
c* Hello World Fortran Example
c*
c* To compile:
c* mpxlf_r -o hello_world_f hello_world.f
c*
c*********************************************************************
c -------------------------------------------------------------------
c  Basic program to demonstrate compilation and execution techniques
c -------------------------------------------------------------------
c     program  hello

implicit none
write(6,*)'Hello, World!'

stop
end
```

To compile these programs, you just invoke the appropriate compiler script:

```
$ mpcc_r -o hello_world_c hello_world.c

$ mpxlf_r -o hello_world_f hello_world.f
** main   === End of Compilation 1 ===
1501-510  Compilation successful for file hello_world.f.
```

POE scripts **mpcc_r**, **mpCC_r**, and **mpxlf_r** link the parallel libraries that allow programs to run in parallel. Script **mpcc_r** generates thread-aware code by linking in the threaded version of MPI, including the threaded POE utility library. Currently, only the threaded version of MPI is provided by POE.

Legacy POE scripts **mpcc**, **mpCC**, and **mpxlf** are now symbolic links to **mpcc_r**, **mpCC_r**, and **mpxlf_r**, and are used in some of the examples.

All the compiler scripts accept all the same options that the non-parallel compilers do, as well as some options specific to POE. For a complete list of all parallel-specific compilation options, see *IBM Parallel Environment: Operation and Use, Volume 1*.

Running one of the POE compiler scripts creates an executable version of the source program that takes advantage of POE. However, before POE can run the program, you need to make sure that it is accessible on each remote node. You can do this by either copying it there, or by mounting the file system that the program resides in to each remote node.

Here is the output of the C program (threaded or non-threaded):

```
$ poe hello_world_c -procs 4

Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

And here is the output of the Fortran program:

```
$ poe hello_world_f -procs 4

Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

POE options

There are a number of options (command line flags) that you may want to specify when invoking POE. These options are covered in full detail in *IBM Parallel Environment: Operation and Use, Volume 1* but here are the ones you will most likely need to be familiar with at this stage.

-procs: When you set **-procs**, you are telling POE how many tasks the program will run. You can also set the **MP_PROCS** environment variable to do this (**-procs** can be used to temporarily override it).

-hostfile or -hfile: The default host list file used by POE to allocate nodes is called *host.list*. You can specify a file other than *host.list* by setting the **-hostfile** or **-hfile** options when invoking POE. You can also set the **MP_HOSTFILE** environment variable to do this (**-hostfile** and **-hfile** can be used to temporarily override it).

-labelio: You can set the **-labelio** option when invoking POE so that the output from the parallel tasks of the program are labeled by task id. This becomes especially useful when you are running a parallel program and the output is *unordered*. When you have output that is labeled output, you can easily determine which message the task returned.

You can also set the **MP_LABELIO** environment variable to do this (**-labelio** can be used to temporarily override it).

-infolevel or -ilevel: You can use the **-infolevel** or **-ilevel** options to specify the level of messages you want from POE. There are different levels of informational, warning, and error messages, plus several debugging levels. The **-infolevel** option generates large amounts of output. Use it with care. You can also set the **MP_INFOLEVEL** environment variable to do this (**-infolevel** and **-ilevel** can be used to temporarily override it).

-pmdlog: The **-pmdlog** option lets you specify that diagnostic messages should be logged to a file on each of the remote nodes of the partition. These diagnostic logs are particularly useful for isolating the cause of abnormal termination. The **-pmdlog** option consumes a significant amount of system resources. Use it with care. You can also set the **MP_PMDLOG** environment variable to do this, and **-pmdlog** can be used to temporarily override the **MP_PMDLOG** environment variable.

The PMD log file is located in **/tmp** and is named: **mplog.***jobid***.***taskid*.

-stdoutmode: The **-stdoutmode** option lets you specify how you want the output data from each task in the program to be displayed. When you set this option to *ordered*, the output data from each parallel task is written to its own buffer, and later, all buffers are flushed, in task order, to STDOUT. The examples here show you how this works. Using the **-infolevel** option consumes a significant amount of system resources, which may affect performance. You can also set the **MP_STDOUTMODE** environment variable to do this (**-stdoutmode** can be used to temporarily override it).

Managing jobs

So far, you have explicitly specified to POE the set of nodes on which to run the parallel application. You did this by creating a list of hosts in a file called *host.list*, in the directory from which you submitted the parallel job. In the absence of any other instructions, POE selected host names out of this file until it had as many as the number of processes you told POE to use (with the **-procs** option).

Another way to tell POE which hosts to use is with LoadLeveler. LoadLeveler can manage jobs on a networked cluster of pSeries servers.

LoadLeveler is a job management system that allows users to run more jobs in less time by matching the jobs' processing needs with the available resources. LoadLeveler allocates nodes, one job at a time. This is necessary if a parallel application is communicating directly over the high performance switch. With the **-euilib** command line option (or the **MP_EUILIB** environment variable), you can specify how you want to do message passing. This option lets you specify the message passing subsystem library implementation, IP or User Space (US), that you wish to use. See *IBM Parallel Environment: Operation and Use, Volume 1* for more information. With LoadLeveler, you can also dedicate the parallel nodes to a single job, so there is no conflict or contention for resources. LoadLeveler allocates nodes from either the host list file, or from a predefined *pool*, which the system administrator usually sets up.

How the nodes are allocated: To know who is allocating the nodes and where they are being allocated from, you must always have a *host list* file or use the **MP_RMPOOL** environment variable or **-rmpool** command line option (unless you are using the **MP_LLFILE** environment variable or the **-llfile** command line option). See *IBM Parallel Environment for AIX: Operation and Use, Volume 1* for more information.

The default for the *host list* file is a file named *host.list* in the directory from which the job is submitted. This default may be overridden by the **-hostfile** command line option or the **MP_HOSTFILE** environment variable. For example, the following command:

```
$ poe hostname -procs 4 -hostfile $HOME/myHosts
```

uses a file called **myHosts**, located in the home directory. If the value of the **-hostfile** parameter does not start with a slash (/), it is taken as relative to the current directory. If the value starts with a slash (/), it is taken as a fully-qualified file name.

For specific examples of how a system administrator defines pools, see *Tivoli® Workload Scheduler LoadLeveler: Using and Administering*. There is another way to designate the pool on which you want the program to run. If **myHosts** did not contain any pool numbers, you could use the:

- **MP_RMPOOL** environment variable which you can set to a number or string. This setting would last for the duration of the session.
- **-rmpool** command line option to specify a number or string when you invoke the program. This option would override the **MP_RMPOOL** environment variable.

If a host list file named **host.list** exists, or if a host list file is specified using **MP_HOSTFILE** or **-hostfile**, anything you specify with **MP_RMPOOL** or **-rmpool** will be ignored. If a file named host.list exists and you want to use **MP_RMPOOL** or **-rmpool** then **MP_HOSTFILE** or **-hostfile** must be set to NULL.

For more information about the **MP_RMPOOL** environment variable or the **-rmpool** command line option, see *IBM Parallel Environment: Operation and Use, Volume 1*.

| If the **myHosts** file contains actual host names, but you want to use the switch
| directly for communication, LoadLeveler allocates only the nodes that are listed in
| **myHosts**. LoadLeveler keeps track of which parallel jobs are using the switch.
| Since it allows more than one job at a time to use the switch, LoadLeveler makes
| sure that only the allowed number of tasks actually use it. If the host list file
| contains actual host names, but you do not want to use the switch directly for
| communication, POE allocates the nodes from those listed in the host list file.

You cannot have both host names and pool IDs in the same host list file.

| The program executes exactly the same way, regardless of whether POE or
| LoadLeveler allocated the nodes. In the following example, the host list file contains
| a pool number which causes the job management system to allocate nodes.
| However, the output is identical to the output in "Compiling" on page 7, where POE
| allocated the nodes from the host list file.

```
$ poe hello_world_c -procs 4 -hostfile pool.list

Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

So, if the output looks the same, regardless of how the nodes are allocated, how do you know whether LoadLeveler was used? Well, POE knows a lot that it ordinarily does not tell you. If you coax it with the **-infolevel** option, POE will tell you more information than you ever wanted to know.

Getting a little more information

You can control the level of messages you get from POE as the program executes by using the **-infolevel** option of POE. The default setting is 1 (normal), which says that warning and error messages from POE will be written to STDERR. However, you can use this option to get more information about how the program executes. For example, with **-infolevel** set to 2, you see a couple of different things. First, you will see a message that says that POE has contacted LoadLeveler. Following that, you will see messages that indicate which nodes LoadLeveler passed back to POE for use.

For a description of the various **-infolevel** settings, see *IBM Parallel Environment: Operation and Use, Volume 1*.

Here is the *hello world* program again:

```
$poe hello_world_c -resd yes -procs 2 -labelio yes -infolevel 2
```

You should see output similar to the following:

```
INFO: 0031-364  Contacting LoadLeveler to set and query information for
      interactive job
INFO: 0031-380  LoadLeveler step ID is k133rp03.kgn.ibm.com.1154.0
INFO: 0031-118  Host k133rp03.kgn.ibm.com requested for task 0
INFO: 0031-118  Host k133rp03.kgn.ibm.com requested for task 1
```

```
INFO: 0031-119  Host k133rp03.kgn.ibm.com allocated for task 0
INFO: 0031-120  Host address 89.116.113.22 allocated for task 0
INFO: 0031-377  Using en0 for mpi euidevice for task 0
INFO: 0031-119  Host k133rp03.kgn.ibm.com allocated for task 1
INFO: 0031-120  Host address 89.116.113.22 allocated for task 1
INFO: 0031-377  Using en0 for mpi euidevice for task 1
   0:INFO: 0031-724  Executing program: <hello_world_c>
   1:INFO: 0031-724  Executing program: <hello_world_c>
   0:Hello, world!
   0:INFO: 0031-306  pm_atexit: pm_exit_value is 0.
   1:Hello, world!
   1:INFO: 0031-306  pm_atexit: pm_exit_value is 0.
INFO: 0031-656  I/O file STDOUT closed by task 0
INFO: 0031-656  I/O file STDERR closed by task 0
INFO: 0031-656  I/O file STDOUT closed by task 1
INFO: 0031-656  I/O file STDERR closed by task 1
INFO: 0031-251  task 0 exited: rc=0
INFO: 0031-251  task 1 exited: rc=0
INFO: 0031-639  Exit status from pm_respond = 0
```

With **-infolevel** set to 2, you also see messages from each node that indicate the executable they are running and what the return code from the executable is. In the example above, you can differentiate between the **-infolevel** messages that come from POE itself and the messages that come from the remote nodes, because the remote nodes are prefixed with their task ID. If you did not set **-infolevel**, you would see only the output of the executable (Hello world!, in the previous example), interspersed with POE output from remote nodes.

With **-infolevel** set to 3, you get more information. In the following example, use the host list file that contains host names again (as opposed to a Pool ID), when you invoke POE.

Look at the following output. In this case, POE tells you that it is opening the host list file, the nodes it found in the file (along with their Internet addresses), the parameters to the executable being run, and the values of some of the POE parameters.

```
$poe hello_world_c -resd yes -procs 2 -labelio yes -ilevel 3
```

You should see output similar to the following:

```
INFO: DEBUG_LEVEL changed from 0 to 1
D1<L1>: Open of file ./host.list successful
D1<L1>: mp_euilib = ip
D1<L1>: 03/04 14:55:13.282519  task 0 k151f1rp02.kgn.ibm.com  10
D1<L1>: 03/04 14:55:13.282677  task 1 k151f1rp02.kgn.ibm.com  10
D1<L1>: node allocation strategy = 2
INFO: 0031-364  Contacting LoadLeveler to set and query information
for interactive job
D1<L1>: 03/04 14:55:13.422268  Calling ll_init_job.

D1<L1>: 03/04 14:55:13.460772  ll_init_job returned.

D1<L1>: 03/04 14:55:13.461426  Job Command String:
#@ job_type = parallel
#@ environment = COPY_ALL
#@ node_usage = shared
#@ bulkxfer = NO
#@ class = No_Class
#@ queue

INFO: 0031-380  LoadLeveler step ID is k151f1rp02.kgn.ibm.com.324.0
INFO: 0031-118  Host k151f1rp02.kgn.ibm.com requested for task 0
INFO: 0031-118  Host k151f1rp02.kgn.ibm.com requested for task 1
INFO: 0031-119  Host k151f1rp02.kgnk.ibm.com allocated for task 0
```

```
INFO: 0031-120  Host address 89.116.157.7 allocated for task 0
INFO: 0031-377  Using en0 for mpi euidevice for task 0
INFO: 0031-119  Host k151f1rp02.kgn.ibm.com allocated for task 1
INFO: 0031-120  Host address 89.116.157.7 allocated for task 1
INFO: 0031-377  Using en0 for mpi euidevice for task 1
D1<L1>: Entering pm_contact, jobid is 0
D1<L1>: Jobid = 1110510899
D1<L1>: Spawning /etc/pmdv4 on all nodes
D1<L1>: 1 master nodes
D1<L1>: 03/04 14:55:15.377008  Calling ll_spawn_connect for node 0,
host name k151f1rp02.kgn.ibm.com.

D1<L1>: TASKID is 0
D1<L1>: 03/04 14:55:15.377576  ll_spawn_connect returned for node 0,
socket fd 6,
host name k151f1rp02.kgn.ibm.com.

D1<L1>: 03/04 14:55:15.377680  Calling pm_spawn_ready.

D1<L1>: 03/04 14:55:15.378916  returned from pm_spawn_ready.

D1<L1>: Socket file descriptor for master 0 (k151f1rp02.kgn.ibm.com) is 6
D1<L1>: SSM_read on socket 6, source = 0, task id: 0, nread: 12, type:3.
D1<L1>: Leaving pm_contact, jobid is 1110510899
D1<L1>: attempting to bind socket to /tmp/s.pedb.544784.1079

   0:INFO: 0031-724  Executing program: <hello_world_c>
   0:D1<L1>: Affinity is not requested; MP_TASK_AFFINITY: -1
   1:INFO: 0031-724  Executing program: <hello_world_c>
   1:D1<L1>: Affinity is not requested; MP_TASK_AFFINITY: -1
   0:INFO: DEBUG_LEVEL changed from 0 to 1
   1:INFO: DEBUG_LEVEL changed from 0 to 1
   0:D1<L1>: In mp_main, mp_main will not be checkpointable
   0:D1<L1>: mp_euilib is <ip>
   0:Hello, World!
   0:INFO: 0031-306  pm_atexit: pm_exit_value is 0.
   1:D1<L1>: In mp_main, mp_main will not be checkpointable
   1:D1<L1>: mp_euilib is <ip>
   1:Hello, World!
   1:INFO: 0031-306  pm_atexit: pm_exit_value is 0.
INFO: 0031-656  I/O file STDOUT closed by task 0
INFO: 0031-656  I/O file STDERR closed by task 0
INFO: 0031-656  I/O file STDOUT closed by task 1
INFO: 0031-656  I/O file STDERR closed by task 1
D1<L1>: Accounting data from task 1 for source 1:
D1<L1>: Accounting data from task 0 for source 0:
INFO: 0031-251  task 0 exited: rc=0
INFO: 0031-251  task 1 exited: rc=0
D1<L1>: All remote tasks have exited: maxx_errcode = 0
INFO: 0031-639  Exit status from pm_respond = 0
D1<L1>: Maximum return code from user = 0
```

The **-infolevel** messages give you more information about what is happening on the home node, but if you want to see what is happening on the remote nodes, you need to use the **-pmdlog** option. If you set **-pmdlog** to a value of *yes*, a log is written to each of the remote nodes that tells you what POE did while running each task.

If you issue the following command, a file is written in **/tmp**, of each remote node, called **mplog.***jobid.taskid*,

```
$ poe hello_world_c -procs 4 -pmdlog yes
```

If **-infolevel** is set to 3 or higher, The job ID will be displayed in the output. If you do not know what the job ID is, it is probably the most recent log file. If you are sharing the node with other POE users, the job ID will be *one* of the most recent log files (but you own the file, so you should be able to tell).

Here is a sample log file. In this example, all four tasks are running on the same node. For more information about how POE runs with multiple tasks on the same node, see Appendix A, "A sample program to illustrate messages," on page 97.

```
AIX Parallel Environment pmd4 version @(#) 2003/06/11 13:19:38
The ID of this process is 520240
The version of this pmd for version checking is 4100
The hostname of this node is k151f1rp02.kgn.ibm.com
The short hostname of this node is k151f1rp02
The taskid of this task is 0
HOMENAME: k151f1rp02.kgn.ibm.com
USERID: 1079
USERNAME: voe3
GROUPID: 100
GROUPNAME: usr
PWD: /u/voe3/pfc/samples/ch01
PRIORITY: 0
NPROCS: 4
PMDLOG: 1
NEWJOB: 0
PDBX: 0
AFSTOKEN: 5765-F83 AIX Parallel Environment
LIBPATH: /usr/lpp/ppe.poe/lib/ip
VERSION (of home node): 4100
JOBID: 1110380176
ENVC recv'd
envc: 31
envc is 31
env[0] = _=hello_world_c
env[1] = MANPATH=/usr/lpp/LoadL/full/man:/usr/lpp/LoadL/so/man
env[2] = LANG=C
env[3] = LOGIN=voe3
env[4] =
PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:/u/voe3/bin:/usr/bin/X11:/sbin:
/usr/local/bin:/usr/lpp/LoadL/full/bin:.
env[5] = LC__FASTMSG=true
env[6] = HISTFILE=/u/voe3/.sh_history/sh_history_307370
env[7] = LOGNAME=voe3
env[8] = MAIL=/usr/spool/mail/voe3
env[9] = LOCPATH=/usr/lib/nls/loc
env[10] = USER=voe3
env[11] = AUTHSTATE=compat
env[12] = SHELL=/bin/ksh
env[13] = ODMDIR=/etc/objrepos
env[14] = HOME=/u/voe3
env[15] = TERM=aixterm
env[16] = MAILMSG=[YOU HAVE NEW MAIL]
env[17] = PWD=/u/voe3/pfc/samples/ch01
env[18] = TZ=EST5EDT
env[19] = ENV=/u/voe3/.kshrc
env[20] = A__z=! LOGNAME
env[21] = NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
env[22] = MP_PROCS=4
env[23] = MP_PMDLOG=YES
env[24] = MP_EUIDEVICE=en0
env[25] = MP_PGMMODEL=SPMD
env[26] = MP_TASK_AFFINITY=-1
env[27] = MP_MSG_API=MPI
env[28] = MP_ISATTY_STDIN=1
env[29] = MP_ISATTY_STDOUT=1
env[30] = MP_ISATTY_STDERR=1
Couldn't open /etc/poe.limits
MASTERS: 1
TASKS: 4:0:1:2:3
```

```
Total number of tasks is 4
Task id for task 1 is 0
Task id for task 2 is 1
Task id for task 3 is 2
Task id for task 4 is 3
TASK_ENV: 0:1 MP_CHILD_INET_ADDR=@1:89.116.107.5,ip 1:1
MP_CHILD_INET_ADDR=@1:9.114.127.2,
ip 2:1 MP_CHILD_INET_ADDR=@1:89.116.107.5,ip 3:1
MP_CHILD_INET_ADDR=@1:89.116.107.5,ip
Number of environment variables is 1
Environment specific data for task 1, task id 0 :
 -- MP_CHILD_INET_ADDR=@1:89.116.107.5,ip
Number of environment variables is 1
Environment specific data for task 2, task id 1 :
 -- MP_CHILD_INET_ADDR=@1:89.116.107.52,ip
Number of environment variables is 1
Environment specific data for task 3, task id 2 :
 -- MP_CHILD_INET_ADDR=@1:89.116.107.52,ip
Number of environment variables is 1
Environment specific data for task 4, task id 3 :
 -- MP_CHILD_INET_ADDR=@1:89.116.107.5,ip
Initial data msg received and parsed
Info level = 1
Doing ruserok() user validation
User validation complete
About to do user root chk
User root check complete
spkeyfuncs not found, continuing....
ident_match not found, continuing....
task information parsed
sb_max used for sndbuf, sndbuf set to 1048576
STDOUT socket SO_SNDBUF set to 1048576
STDOUT socket SO_RCVBUF set to 67424
main thread id is 1 before Setup signal handler for termination.
newjob is 0.
msg read, type is 13
string = <hello_world_c> SSM_CMD_STR recv'd
command string is <hello_world_c>
0: pm_putargs: argc = 1, k = 1
1: pm_putargs: argc = 1, k = 1
2: pm_putargs: argc = 1, k = 1
3: pm_putargs: argc = 1, k = 1
SSM_CMD_STR parsed
SSM_EXT_DEBUG msg, type is 46
child pipes created
Task 0 OS version 5 , release 2
child: pipes successfully duped for task 0
0: MP_COMMON_TASKS is <3:1:2:3>
0: partition id is <1110380176>
Task 1 OS version 5 , release 2
child: pipes successfully duped for task 1
1: MP_COMMON_TASKS is <3:0:2:3>
1: partition id is <1110380176>
after initgroups (*group_struct).gr_gid = 100
after initgroups (*group_struct).gr_name = usr
Task 2 OS version 5 , release 2
child: pipes successfully duped for task 2
2: MP_COMMON_TASKS is <3:0:1:3>
2: partition id is <1110380176>
pmd child: core limit is 9223372036854775807,
hard limit is 9223372036854775807
pmd child: rss limit is 9223372036854775807,
hard limit is 9223372036854775807
pmd child: stack limit is 9223372036854775807,
hard limit is 9223372036854775807
pmd child: data segment limit is 9223372036854775807,
hard limit is 9223372036854775807
```

```
pmd child: cpu time limit is 9223372036854775807,
hard limit is 9223372036854775807
pmd child: file size limit is 9223372036854775807,
hard limit is 9223372036854775807
0: (*group_struct).gr_gid = 100
0: (*group_struct).gr_name = usr
0: userid, groupid and cwd set!
0: current directory is /u/voe3/pfc/samples/ch01
0: about to start the user's program
0: argument list:
argv[0] for task 0 = hello_world_c

argv[1] (in hex) = 0
child: environment for task 0:
    task_env[0] = MP_CHILD_INET_ADDR=@1:89.116.107.5,ip

child: common environment data for all tasks:
    env[0] = _=hello_world_c
    env[1] = MANPATH=/usr/lpp/LoadL/full/man:/usr/lpp/LoadL/so/man
    env[2] = LANG=C
    env[3] = LOGIN=voe3
    env[4] = PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:/u/voe3/bin:
/usr/bin/X11:/sbin:/usr/local/bin:/usr/lpp/LoadL/full/bin:.
    env[5] = LC__FASTMSG=true
    env[6] = HISTFILE=/u/voe3/.sh_history/sh_history_307370
    env[7] = LOGNAME=voe3
    env[8] = MAIL=/usr/spool/mail/voe3
    env[9] = LOCPATH=/usr/lib/nls/loc
    env[10] = USER=voe3
    env[11] = AUTHSTATE=compat
    env[12] = SHELL=/bin/ksh
    env[13] = ODMDIR=/etc/objrepos
    env[14] = HOME=/u/voe3
    env[15] = TERM=aixterm
    env[16] = MAILMSG=[YOU HAVE NEW MAIL]
    env[17] = PWD=/u/voe3/pfc/samples/ch01
    env[18] = TZ=EST5EDT
    env[19] = ENV=/u/voe3/.kshrc
    env[20] = A__z=! LOGNAME
    env[21] = NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
    env[22] = MP_PROCS=4
    env[23] = MP_PMDLOG=YES
    env[24] = MP_EUIDEVICE=en0
    env[25] = MP_PGMMODEL=SPMD
    env[26] = MP_TASK_AFFINITY=-1
    env[27] = MP_MSG_API=MPI
    env[28] = MP_ISATTY_STDIN=1
    env[29] = MP_ISATTY_STDOUT=1
    env[30] = MP_ISATTY_STDERR=1

0: LIBPATH = /usr/lpp/ppe.poe/lib/ip
MP_TASK_AFFINITY = -1, MP_MPI_NETWORK =
Affinity is not requested
parent: task 0 forked, child pid is 524460
Task 3 OS version 5 , release 2
child: pipes successfully duped for task 3
3: MP_COMMON_TASKS is <3:0:1:2>
3: partition id is <1110380176>
attach data sent for task 0
parent: task 1 forked, child pid is 549054
attach data sent for task 1
parent: task 2 forked, child pid is 516106
attach data sent for task 2
1: (*group_struct).gr_gid = 100
1: (*group_struct).gr_name = usr
parent: task 3 forked, child pid is 417868
attach data sent for task 3
```

```
1: userid, groupid and cwd set!
1: current directory is /u/voe3/pfc/samples/ch01
1: about to start the user's program
1: argument list:
argv[0] for task 1 = hello_world_c

argv[1] (in hex) = 0
child: environment for task 1:
    task_env[0] = MP_CHILD_INET_ADDR=@1:89.116.107.5,ip

1: LIBPATH = /usr/lpp/ppe.poe/lib/ip
MP_TASK_AFFINITY = -1, MP_MPI_NETWORK =
Affinity is not requested
2: (*group_struct).gr_gid = 100
2: (*group_struct).gr_name = usr
2: userid, groupid and cwd set!
2: current directory is /u/voe3/pfc/samples/ch01
2: about to start the user's program
2: argument list:
argv[0] for task 2 = hello_world_c

argv[1] (in hex) = 0
child: environment for task 2:
    task_env[0] = MP_CHILD_INET_ADDR=@1:89.116.107.5,ip

2: LIBPATH = /usr/lpp/ppe.poe/lib/ip
MP_TASK_AFFINITY = -1, MP_MPI_NETWORK =
Affinity is not requested
3: (*group_struct).gr_gid = 100
3: (*group_struct).gr_name = usr
3: userid, groupid and cwd set!
3: current directory is /u/voe3/pfc/samples/ch01
3: about to start the user's program
3: argument list:
argv[0] for task 3 = hello_world_c

argv[1] (in hex) = 0
child: environment for task 3:
    task_env[0] = MP_CHILD_INET_ADDR=@1:89.116.107.5,ip

3: LIBPATH = /usr/lpp/ppe.poe/lib/ip
MP_TASK_AFFINITY = -1, MP_MPI_NETWORK =
Affinity is not requested
select: rc = 1
pulse is on, curr_time is 1109966440, send_time is 0, select time is 600
pulse sent at 1109966440 count is 0
0: pmd parent: cntl pipe read OK:
0: pmd parent: type: 26, srce: 0, dest: -2, bytes: 7
parent: SSM_CHILD_PID: 524460
select: rc = 3
pulse is on, curr_time is 1109966440, send_time is 1109966440, select time is 600
in pmd select, SSM_read ok, SSM_type=34.
pulse received at 1109966440 received count is 0
pmd parent: STDOUT read OK for task 0
0: STDOUT: Hello, World!

0: pmd parent: cntl pipe read OK:
0: pmd parent: type: 47, srce: 0, dest: -2, bytes: 5
parent: childs version is 4100.
parent: home node version is 4100.
parent: this pmd version is 4100.
select: rc = 1
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
0: pmd parent: cntl pipe read OK:
0: pmd parent: type: 17, srce: 0, dest: -1, bytes: 2
select: rc = 1
```

```
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
1: pmd parent: cntl pipe read OK:
1: pmd parent: type: 26, srce: 1, dest: -2, bytes: 7
parent: SSM_CHILD_PID: 549054
select: rc = 2
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
pmd parent: STDOUT read OK for task 1
1: STDOUT: Hello, World!

1: pmd parent: cntl pipe read OK:
1: pmd parent: type: 47, srce: 1, dest: -2, bytes: 5
parent: childs version is 4100.
parent: home node version is 4100.
parent: this pmd version is 4100.
select: rc = 1
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
1: pmd parent: cntl pipe read OK:
1: pmd parent: type: 17, srce: 1, dest: -1, bytes: 2
select: rc = 1
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
2: pmd parent: cntl pipe read OK:
2: pmd parent: type: 26, srce: 2, dest: -2, bytes: 7
parent: SSM_CHILD_PID: 516106
select: rc = 2
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
pmd parent: STDOUT read OK for task 2
2: STDOUT: Hello, World!

2: pmd parent: cntl pipe read OK:
2: pmd parent: type: 47, srce: 2, dest: -2, bytes: 5
parent: childs version is 4100.
parent: home node version is 4100.
parent: this pmd version is 4100.
select: rc = 1
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
2: pmd parent: cntl pipe read OK:
2: pmd parent: type: 17, srce: 2, dest: -1, bytes: 2
select: rc = 1
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
3: pmd parent: cntl pipe read OK:
3: pmd parent: type: 26, srce: 3, dest: -2, bytes: 7
parent: SSM_CHILD_PID: 417868
select: rc = 2
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
pmd parent: STDOUT read OK for task 3
3: STDOUT: Hello, World!

3: pmd parent: cntl pipe read OK:
3: pmd parent: type: 47, srce: 3, dest: -2, bytes: 5
parent: childs version is 4100.
parent: home node version is 4100.
parent: this pmd version is 4100.
select: rc = 1
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
3: pmd parent: cntl pipe read OK:
3: pmd parent: type: 17, srce: 3, dest: -1, bytes: 2
select: rc = 1
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
in pmd select, SSM_read ok, SSM_type=5.
select: rc = 5
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
pmd send SSM_IO_CLOSED to poe for stdout_open
pmd send SSM_IO_CLOSED to poe for stdout_open
pmd send SSM_IO_CLOSED to poe for stdout_open
pmd send SSM_IO_CLOSED to poe for stdout_open
3: count = 0 on stderr
pmd send SSM_IO_CLOSED to poe for stderr_open
select: rc = 7
pulse is on, curr_time is 1109966440, send_time is 1109966440,
select time is 600
0: count = 0 on stderr
pmd send SSM_IO_CLOSED to poe for stderr_open
1: count = 0 on stderr
pmd send SSM_IO_CLOSED to poe for stderr_open
2: count = 0 on stderr
pmd send SSM_IO_CLOSED to poe for stderr_open
in pmd signal handler for task 3, signal 20
3: wait status is 00000000
Exiting child for task 3, PID: 417868
err_data for task 3 is 0
2: wait status is 00000000
Exiting child for task 2, PID: 516106
err_data for task 2 is 0
1: wait status is 00000000
Exiting child for task 1, PID: 549054
err_data for task 1 is 0
0: wait status is 00000000
Exiting child for task 0, PID: 524460
err_data for task 0 is 0
in pmd signal handler, wait returned -1...
parent: child exited and all pipes closed for all tasks
err_data for task 0 is 0
err_data for task 1 is 0
err_data for task 2 is 0
err_data for task 3 is 0
pmd_exit reached!, exit code is 0
No collective communication shared memory segments to clean up.
```

Appendix A, "A sample program to illustrate messages," on page 97 includes an example of setting **-infolevel** to 6, and explains the important lines of output.

Chapter 2. Message passing

If you are familiar with message passing parallel programming, and you are familiar with message passing protocols, you can skip ahead to Chapter 3, "Diagnosing and correcting common problems," on page 35 for a discussion on using the PE tools. If you are familiar with message passing parallel programming, but you would like to know more about the PE message passing protocols, look at the information in "Protocols supported" on page 32.

This is a discussion of some of the techniques for creating a parallel program, using message passing, and the various advantages and pitfalls associated with each technique. It does not provide an in-depth tutorial on writing parallel programs. Instead, it is an introduction to basic message passing parallel concepts.

To create a successful parallel program, start with a working sequential program. Complex sequential programs are difficult to get working correctly, without also having to worry about the additional complexity introduced by parallelism and message passing. It is easier to convert a working serial program to parallel than it is to create a parallel program from scratch. As you become proficient at creating parallel programs, you will develop an awareness of which sequential techniques translate better into parallel implementations. Once aware, you can then make a point of using these techniques in your sequential programs. This is a discussion of some of the fundamentals of creating parallel programs.

There are two common techniques for turning a sequential program into a parallel program; *data decomposition* and *functional decomposition*. Data decomposition means distributing the data that the program is processing among the parallel tasks. Each parallel task does approximately the same thing but on a different set of data. With functional decomposition, the function that the application is performing is distributed among the tasks. Each task operates on the same data, but does something different. Most parallel programs do not use data decomposition or functional decomposition exclusively. Rather, they use a mixture of the two, weighted more toward one type or the other. One way to implement either form of decomposition is through the use of message passing.

The message passing model

The message passing model of communication is typically used in distributed memory systems, where each processor node owns private memory, and is linked by an interconnection network. The high performance switch provides the interconnection network needed for high-speed exchange of messages. With message passing, each task operates exclusively in a private environment, but must cooperate with other tasks to interact. In this situation, tasks must exchange messages to interact with one another.

The challenge of the message passing model is in reducing message traffic over the interconnection network while ensuring that the correct and updated values of the passed data are promptly available to the tasks, when required. Optimizing message traffic boosts performance.

Synchronization is the act of forcing events to occur at the same time or in a certain order. Synchronization requires taking into account the logical dependence and the order of precedence among the tasks. You can describe the message passing model as self-synchronizing because the mechanism of sending and receiving messages involves implicit synchronization points. To put it another way, a message cannot be received if it has not already been sent.

Data decomposition

A good technique for making a sequential application parallel is to look for loops where each iteration does not depend on any prior iteration (this is also a prerequisite for either *unrolling* or eliminating loops). An example of a loop that has dependencies on prior iterations is the loop for computing the Factorial series. The value calculated by each iteration depends on the value resulting from the previous pass. If each iteration of a loop does not depend on a previous iteration, the data being processed can be processed in parallel, with two or more iterations being performed simultaneously.

The C program example below includes a loop with independent iterations. This example does not include the routines for computing the coefficient and determinant because they are not part of the parallelization at this point.

```
/************************************************************************
*
* Matrix Inversion Program - serial version
*
* To compile:
* cc -o inverse_serial inverse_serial.c
*
*************************************************************************/

#include<stdlib.h>
#include<stdio.h>
#include<assert.h>
#include<errno.h>

float determinant(float **matrix,
    int size,
    int * used_rows,
    int * used_cols,
    int depth);
float coefficient(float **matrix,int size, int row, int col);
void print_matrix(FILE * fptr,float ** mat,int rows, int cols);
float test_data[8][8] =  {
    {4.0, 2.0, 4.0, 5.0, 4.0, -2.0, 4.0, 5.0},
    {4.0, 2.0, 4.0, 5.0, 3.0, 9.0, 12.0, 1.0 },
    {3.0, 9.0, -13.0, 15.0, 3.0, 9.0, 12.0, 15.0},
    {3.0, 9.0, 12.0, 15.0, 4.0, 2.0, 7.0, 5.0 },
    {2.0, 4.0, -11.0, 10.0, 2.0, 4.0, 11.0, 10.0 },
    {2.0, 4.0, 11.0, 10.0, 3.0, -5.0, 12.0, 15.0 },
    {1.0, -2.0, 4.0, 10.0, 3.0, 9.0, -12.0, 15.0 } ,
    {1.0, 2.0, 4.0, 10.0, 2.0, -4.0, -11.0, 10.0 } ,
  };
#define ROWS 8

int main(int argc, char **argv)
{

  float **matrix;
  float **inverse;
  int   rows,i,j;
  float determ;
  int * used_rows, * used_cols;

  rows = ROWS;

  /* Allocate markers to record rows and columns to be skipped */
  /* during determinant calculation                            */
  used_rows = (int *)    malloc(rows*sizeof(*used_rows));
  used_cols = (int *)    malloc(rows*sizeof(*used_cols));

  /* Allocate working copy of matrix and initialize it from static copy */
  matrix  =   (float **) malloc(rows*sizeof(*matrix));
  inverse =   (float **) malloc(rows*sizeof(*inverse));
  for(i=0;i<rows;i++)
    {
      matrix[i]  = (float *) malloc(rows*sizeof(**matrix));
      inverse[i] = (float *) malloc(rows*sizeof(**inverse));
      for(j=0;j<rows;j++)
          matrix[i][j] = test_data[i][j];
    }

  /* Compute and print determinant */
  printf("The determinant of\n\n");
  print_matrix(stdout,matrix,rows,rows);
  determ=determinant(matrix,rows,used_rows,used_cols,0);
  printf("\nis %f\n",determ);
  fflush(stdout);
  assert(determ!=0);

  for(i=0;i<rows;i++)
    {
      for(j=0;j<rows;j++)
        {
          inverse[j][i] = coefficient(matrix,rows,i,j)/determ;
        }
    }

  printf("The inverse is\n\n");
  print_matrix(stdout,inverse,rows,rows);

  return (0);
}
```

Before talking about making the algorithm parallel, look at what is necessary to create the program with PE. The example below shows the same program, but it is now aware of PE. You do this by using three calls in the beginning of the routine, and one at the end.

The first of these calls (**MPI_Init**) initializes the *MPI* environment, and the last call (**MPI_Finalize**) closes the environment. **MPI_Comm_size** sets the variable **tasks** to the total number of parallel tasks running this application, and **MPI_Comm_rank** sets **me** to the task ID of the particular instance of the parallel code that invoked it.

MPI_Comm_size actually gets the size of the communicator you pass in and **MPI_COMM_WORLD** is a predefined communicator that includes everybody. For more information about these calls, *IBM Parallel Environment: MPI Subroutine Reference* or other MPI publications may be of some help.

```
/************************************************************************
*
* Matrix Inversion Program - serial version enabled for parallel environment
*
* To compile:
* mpcc -g -o inverse_parallel_enabled inverse_parallel_enabled.c
*
************************************************************************/

#include<stdlib.h>
#include<stdio.h>
#include<assert.h>
#include<errno.h>
#include<mpi.h>
```

```
float determinant(float **matrix,int size, int * used_rows, int * used_cols,
                  int depth);
float coefficient(float **matrix,int size, int row, int col);
void print_matrix(FILE * fptr,float ** mat,int rows, int cols);
float test_data[8][8] =  {
      {4.0, 2.0, 4.0, 5.0, 4.0, -2.0, 4.0, 5.0},
      {4.0, 2.0, 4.0, 5.0, 3.0, 9.0, 12.0, 1.0 },
      {3.0, 9.0, -13.0, 15.0, 3.0, 9.0, 12.0, 15.0},
      {3.0, 9.0, 12.0, 15.0, 4.0, 2.0, 7.0, 5.0 },
      {2.0, 4.0, -11.0, 10.0, 2.0, 4.0, 11.0, 10.0 },
      {2.0, 4.0, 11.0, 10.0, 3.0, -5.0, 12.0, 15.0 },
      {1.0, -2.0, 4.0, 10.0, 3.0, 9.0, -12.0, 15.0 } ,
      {1.0, 2.0, 4.0, 10.0, 2.0, -4.0, -11.0, 10.0 } ,
};
#define ROWS 8

int me, tasks, tag=0;

int main(int argc, char **argv)
{

  float **matrix;
  float **inverse;
  int rows,i,j;
  float determ;
  int * used_rows, * used_cols;

  MPI_Status status[ROWS];   /* Status of messages */
  MPI_Request req[ROWS];  /* Message IDs */

  MPI_Init(&argc,&argv);   /* Initialize MPI */
  MPI_Comm_size(MPI_COMM_WORLD,&tasks); /* How many parallel tasks are there?*/
  MPI_Comm_rank(MPI_COMM_WORLD,&me); /* Who am I? */

  rows = ROWS;

  /* Allocate markers to record rows and columns to be skipped */
  /* during determinant calculation                            */
  used_rows = (int *)    malloc(rows*sizeof(*used_rows));
  used_cols = (int *)    malloc(rows*sizeof(*used_cols));

  /* Allocate working copy of matrix and initialize it from static copy */
  matrix  =   (float **) malloc(rows*sizeof(*matrix));
  inverse =   (float **) malloc(rows*sizeof(*inverse));
  for(i=0;i<rows;i++)
    {
      matrix[i]  = (float *) malloc(rows*sizeof(**matrix));
      inverse[i] = (float *) malloc(rows*sizeof(**inverse));
      for(j=0;j<rows;j++)
      matrix[i][j] = test_data[i][j];
    }

  /* Compute and print determinant */
  printf("The determinant of\n\n");
  print_matrix(stdout,matrix,rows,rows);
  determ=determinant(matrix,rows,used_rows,used_cols,0);
  printf("\nis %f\n",determ);
  fflush(stdout);

  for(i=0;i<rows;i++)
    {
      for(j=0;j<rows;j++)
        {
          inverse[j][i] = coefficient(matrix,rows,i,j)/determ;
        }
    }
  printf("The inverse is\n\n");
  print_matrix(stdout,inverse,rows,rows);

  /* Wait for all parallel tasks to get here, then quit */
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();

  exit(0);
}

float determinant(float **matrix,int size, int * used_rows, int * used_cols,
                  int depth)
  {
    int col1, col2, row1, row2;
    int j,k;
    float total=0;
    int sign = 1;

    /* Find the first unused row */
    for(row1=0;row1<size;row1++)
      {
        for(k=0;k<depth;k++)
          {
            if(row1==used_rows[k]) break;
          }
        if(k>=depth)  /* this row is not used */
          break;
      }
    assert(row1<size);

    if(depth==(size-2))
      {
/* There are only 2 unused rows/columns left */

/* Find the second unused row */
for(row2=row1+1;row2<size;row2++)
  {
    for(k=0;k<depth;k++)
      {
        if(row2==used_rows[k]) break;
      }
    if(k>=depth)  /* this row is not used */
      break;
  }
assert(row2<size);

/* Find the first unused column */
for(col1=0;col1<size;col1++)
  {
    for(k=0;k<depth;k++)
      {
        if(col1==used_cols[k]) break;
      }
    if(k>=depth)  /* this column is not used */
      break;
  }
assert(col1<size);

/* Find the second unused column */
for(col2=col1+1;col2<size;col2++)
  {
    for(k=0;k<depth;k++)
      {
        if(col2==used_cols[k]) break;
      }
    if(k>=depth)  /* this column is not used */
```

```
      break;
  }
assert(col2<size);

/* Determinant = m11*m22-m12*m21 */
return matrix[row1][col1]*matrix[row2][col2]
-matrix[row2][col1]*matrix[row1] [col2];
     }

   /* There are more than 2 rows/columns in the matrix being processed  */
   /* Compute the determinant as the sum of the product of each element */
   /* in the first row and the determinant of the matrix with its row   */
   /* and column removed                                                */
   total = 0;

   used_rows[depth] = row1;
   for(col1=0;col1<size;col1++)
     {
       for(k=0;k<depth;k++)
         {
           if(col1==used_cols[k]) break;
         }
   if(k<depth)  /* This column is used */
     continue;
   used_cols[depth] = col1;
  total += sign*matrix[row1][col1]*determinant(matrix,size,
 used_rows,used_cols,depth+1);
      sign=(sign==1)?-1:1;
    }
  return total;
 }

void print_matrix(FILE * fptr,float ** mat,int rows, int cols)
{
  int i,j;
  for(i=0;i<rows;i++)
    {
      for(j=0;j<cols;j++)
        {
          fprintf(fptr,"%10.4f ",mat[i][j]);
        }
      fprintf(fptr,"\n");
    }
  fflush(fptr);
}

float coefficient(float **matrix,int size, int row, int col)
{
  float coef;
  int * ur, *uc;

  ur = malloc(size*sizeof(matrix));
  uc = malloc(size*sizeof(matrix));
  ur[0]=row;
  uc[0]=col;
  coef = (((row+col)%2)?-1:1)*determinant(matrix,size,ur,uc,1);
  return coef;
}
```

| In this particular example **Matrix Inversion Program - serial version enabled for parallel environment** each parallel task is going to determine the entire inverse matrix, and they are all going to print it out. In the program **Matrix Inversion Program - serial version**, the output of all the tasks is intermixed, so it is difficult to figure out what the answer really is.

A better approach is to distribute the work among several parallel tasks and collect the results when they are done. In this example, the loop that computes the elements of the inverse matrix simply goes through the elements of the inverse matrix, computes the coefficient, and divides it by the determinant of the matrix. Since there is no relationship between elements of the inverse matrix, they can all be computed in parallel.

Every communication call has an associated cost, so you need to balance the benefit of parallelism with the cost of communication. If you were to totally parallelize the inverse matrix element computation, each element would be derived by a separate task. The cost of collecting those individual values back into the inverse matrix would be significant. It might also outweigh the benefit of having reduced the computation cost and time by running the job in parallel. So, instead, you are going to compute the elements of each row in parallel, and send the values back, one row at a time. This way you spread some of the communication overhead over several data values. In this case, you will execute loop 1 in parallel in this next example.

```
************************************************************************
*
* Matrix Inversion Program - First parallel implementation
* To compile:
* mpcc -g -o inverse_parallel inverse_parallel.c
*
************************************************************************

#include<stdlib.h>
#include<stdio.h>
#include<assert.h>
#include<errno.h>
#include<mpi.h>
float determinant(float **matrix,int size, int * used_rows,
                  int * used_cols, int depth);
float coefficient(float **matrix,int size, int row, int col);
void print_matrix(FILE * fptr,float ** mat,int rows, int cols);

float test_data[8][8] =  {
    {4.0, 2.0, 4.0, 5.0, 4.0, -2.0, 4.0, 5.0},
    {4.0, 2.0, 4.0, 5.0, 3.0, 9.0, 12.0, 1.0 },
    {3.0, 9.0, -13.0, 15.0, 3.0, 9.0, 12.0, 15.0},
    {3.0, 9.0, 12.0, 15.0, 4.0, 2.0, 7.0, 5.0 },
    {2.0, 4.0, -11.0, 10.0, 2.0, 4.0, 11.0, 10.0 },
    {2.0, 4.0, 11.0, 10.0, 3.0, -5.0, 12.0, 15.0 },
    {1.0, -2.0, 4.0, 10.0, 3.0, 9.0, -12.0, 15.0 },
    {1.0, 2.0, 4.0, 10.0, 2.0, -4.0, -11.0, 10.0 } ,

};
#define ROWS 8
int me, tasks, tag=0;

int main(int argc, char **argv)
{

  float **matrix;
  float **inverse;
  int rows,i,j;
  float determ;
  int * used_rows, * used_cols;


  MPI_Status status[ROWS];  /* Status of messages */
  MPI_Request req[ROWS];  /* Message IDs */

  MPI_Init(&argc,&argv);  /* Initialize MPI */
```

```
MPI_Comm_size(MPI_COMM_WORLD,&tasks);  /* How many parallel tasks are there?*/
MPI_Comm_rank(MPI_COMM_WORLD,&me);  /* Who am I? */

rows = ROWS;

/* You need exactly one task for each row of the matrix plus one task */
/* to act as coordinator.  If you didn't have this, the last task      */
/* reports the error (so everybody doesn't put out the same message  */
if(tasks!=rows+1)
  {
    if(me==tasks-1)
fprintf(stderr,"%d tasks required for this demo"
"(one more than the number of rows in matrix\n",rows+1)";
     exit(-1);
   }
/* Allocate markers to record rows and columns to be skipped */
/* during determinant calculation                            */
used_rows = (int *)    malloc(rows*sizeof(*used_rows));
used_cols = (int *)    malloc(rows*sizeof(*used_cols));

/* Allocate working copy of matrix and initialize it from static copy */
matrix = (float **) malloc(rows*sizeof(*matrix));
for(i=0;i<rows;i++)
 {
  matrix[i] = (float *) malloc(rows*sizeof(**matrix));
  for(j=0;j<rows;j++)
   matrix[i][j] = test_data[i][j];
 }

/* Everyone computes the determinant (to avoid message transmission) */
determ=determinant(matrix,rows,used_rows,used_cols,0);

if(me==tasks-1)
  {/* The last task acts as coordinator */
   inverse = (float**) malloc(rows*sizeof(*inverse));
   for(i=0;i<rows;i++)
     {
      inverse[i] = (float *) malloc(rows*sizeof(**inverse));
     }
   /* Print the determinant */
   printf("The determinant of\n\n");
   print_matrix(stdout,matrix,rows,rows);
   printf("\nis %f\n",determ);
   /* Collect the rows of the inverse matrix from the other tasks */
   /* First, post a receive from each task into the appropriate row */
   for(i=0;i<rows;i++)
     }
      MPI_Irecv(inverse[i],rows,MPI_REAL,i,tag,MPI_COMM_WORLD,&(req[i]));
     }
   /* Then wait for all the receives to complete */
   MPI_Waitall(rows,req,status);
   printf("The inverse is\n\n");
   print_matrix(stdout,inverse,rows,rows);
  }
 else
  {/* All the other tasks compute a row of the inverse matrix */
   int dest = tasks-1;
   float *one_row;
   int size = rows*sizeof(*one_row);

   one_row = (float*) malloc(size);
   for(j=0;j<rows;j++)
     {
       one_row[j] = coefficient(matrix,rows,j,me)/determ;
     }
   /* Send the row back to the coordinator */
   MPI_Send(one_row,rows,MPI_REAL,dest,tag,MPI_COMM_WORLD);
```

```
    }
/* Wait for all parallel tasks to get here, then quit */
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
}
exit(0);
```

Functional decomposition

Parallel servers and data mining applications are examples of functional decomposition. With functional decomposition, the function that the application is performing is distributed among the tasks. Each task operates on the same data, but does something different. The sine series algorithm is also an example of functional decomposition. With this algorithm, the work being done by each task is trivial. The cost of distributing data to the parallel tasks could outweigh the value of running the program in parallel, and parallelism would increase total time. Another approach to parallelism is to invoke different functions, each of which processes all of the data simultaneously. This is possible as long as the final or intermediate results of any function are not required by another function. For example, searching a matrix for the largest and smallest values as well as a specific value could be done in parallel.

This is a simple example, but suppose the elements of the matrix were arrays of polynomial coefficients. Further, suppose the search involved actually evaluating different polynomial equations using the same coefficients. In this case, it would make sense to evaluate each equation separately.

On a simpler scale, let us look at the series for the sine function:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \frac{x^{2n+1}}{(2n+1)!}$$

The serial approach to solving this problem is to loop through the number of terms desired, accumulating the factorial value and the sine value. When the appropriate number of terms has been computed, the loop exits. The following example does exactly this. In this example, you have an array of values for which you want the sine, and an outer loop would repeat this process for each element of the array. Since you do not want to recompute the factorial each time, you need to allocate an array to hold the factorial values and compute them outside the main loop.

```
/************************************************************************
*
* Series Evaluation - serial version
*
* To compile:
* cc -o series_serial series_serial.c -lm
*
************************************************************************/

#include<stdlib.h>
#include<stdio.h>
#include<math.h>

double angle[] = { 0.0, 0.1*M_PI, 0.2*M_PI, 0.3*M_PI, 0.4*M_PI,
       0.5*M_PI, 0.6*M_PI, 0.7*M_PI, 0.8*M_PI, 0.9*M_PI, M_PI };

#define TERMS 8

int main(int argc, char **argv)
```

```
{
  double divisor[TERMS], sine;
  int a, t, angles = sizeof(angle)/sizeof(angle[0]);

  /* Initialize denominators of series terms */
  divisor[0] = 1;
  for(t=1;t<TERMS;t++)
    {
      divisor[t] = -2*t*(2*t+1)*divisor[t-1];
    }

  /* Compute sine of each angle */
  for(a=0;a<angles;a++)
    {
      sine = 0;
      /* Sum the terms of the series */
      for(t=0;t<TERMS;t++)
        {
          sine += pow(angle[a],(2*t+1))/divisor[t];
        }
      printf("sin(%lf) + %lf\n",angle[a],sine);
        }
}
```

In a parallel environment, you could assign each term to one task and just accumulate the results on a separate node. In fact, that is what the following example does.

```
/************************************************************************
*
* Series Evaluation - parallel version
*
* To compile:
* mpcc -g -o series_parallel series_parallel.c -lm
*
***********************************************************************/

#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<mpi.h>

double angle[] = { 0.0, 0.1*M_PI, 0.2*M_PI, 0.3*M_PI, 0.4*M_PI,
       0.5*M_PI, 0.6*M_PI, 0.7*M_PI, 0.8*M_PI, 0.9*M_PI, M_PI };

int main(int argc, char **argv)
{
  double data, divisor, partial, sine;
  int a, t, angles = sizeof(angle)/sizeof(angle[0]);
  int me, tasks, term;

  MPI_Init(&argc,&argv);    /* Initialize MPI */
  MPI_Comm_size(MPI_COMM_WORLD,&tasks); /* How many parallel tasks are there?*/
  MPI_Comm_rank(MPI_COMM_WORLD,&me);   /* Who am I? */

  term = 2*me+1;   /* Each task computes a term */
  /* Scan the factorial terms through the group members   */
  /* Each member will effectively multiply the product of */
  /* the result of all previous members by its factorial  */
  /* term, resulting in the factorial up to that point    */
  if(me==0)
    data = 1.0;
  else
    data = -(term-1)*term;
  MPI_Scan(&data,&divisor,1,MPI_DOUBLE,MPI_PROD,MPI_COMM_WORLD);

  /* Compute sine of each angle */
  for(a=0;a<angles;a++)
    {
      partial = pow(angle[a],term)/divisor;
      /* Pass all the partials back to task 0 and   */
      /* accumulate them with the MPI_SUM operation */
      MPI_Reduce(&partial,&sine,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
      /* The first task has the total value */
      if(me==0)
        {
          printf("sin(%lf) + %lf\n",angle[a],sine);
        }
    }
  MPI_Finalize();
}
```

With this approach, each task *i* uses its position in the **MPI_COMM_WORLD** communicator group to compute the value of one term. It first computes its working value as 2*i*+1 and calculates the factorial of this value. Since (2*i*+1)! is (2*i*-1)! x 2*i* x (2*i*+1), if each task could get the factorial value computed by the previous task, all it would have to do is multiply it by 2*i* x (2*i*+1). Fortunately, MPI provides the capability to do this with the **MPI_SCAN** function. When **MPI_SCAN** is invoked on the first task in a communication group, the result is the input data to **MPI_SCAN**. When **MPI_SCAN** is invoked on subsequent members of the group, the result is obtained by invoking a function on the result of the previous member of the group and its input data.

The MPI standard is documented in *MPI: A Message-Passing Interface Standard, Version 1.1* and is extended in *MPI: A Message-Passing Interface Standard, Version 2.0*, both of which are available from the University of Tennessee. The standard does not specify how to implement the scan function, so a particular implementation does not have to obtain the result from one task and pass it on to the next for processing. This is, however, a convenient way of visualizing the scan function, and the remainder of the discussion will assume that this is happening.

In the example, the function invoked is the built-in multiplication function, **MPI_PROD**. Task 0 (which is computing 1!) sets its result to 1. Task 2 is computing 3! which it obtains by multiplying 2 x 3 by 1! (the result of Task 0). Task 3 multiplies 3! (the result of Task 2) by 4 to get 4!. This continues until all the tasks have computed their factorial values. The input data to the **MPI_SCAN** calls is made negative so the signs of the divisors will alternate between plus and minus.

Once the divisor for a term has been computed, the loop through all the angles (theta) can be done. The partial term is computed as:

$$\pm \frac{\theta^n}{n!}$$

Then, **MPI_REDUCE** is called which is similar to **MPI_SCAN** except that instead of calling a function on each task, the tasks send their raw data to Task 0, which invokes the function on all data values. The function being invoked in the example is **MPI_SUM** which just adds the data values from all of the tasks. Then, Task 0 prints out the result.

Duplication versus redundancy

In the matrix inversion program, each task goes through the process of allocating the matrix and copying the initialization data into it. So why does not one task do this and send the result to all the other tasks? This example has a trivial initialization process, but in a situation where initialization requires complex time-consuming calculations, this question is even more important.

To understand the answer to this question and, more importantly, be able to apply the understanding to answering the question for other applications, you need to stop and consider the application as a whole. If one task of a parallel application takes on the role of initializer, two things happen. First, all of the other tasks must wait for the initializer to complete (assuming that no work can be done until initialization is completed). Second, some sort of communication must occur to get the results of initialization distributed to all the other tasks. This not only means that there is nothing for the other tasks to do while one task is doing the initializing, there is also a cost associated with sending the results out. Although replicating the initialization process on each of the parallel tasks seems like unnecessary duplication, it allows the tasks to start processing more quickly because they do not have to wait to receive the data.

So, should all initialization be done in parallel? Not necessarily. If the initialization is just computation and setup based on input parameters, each parallel task can initialize independently. Although this seems counter-intuitive at first, because the effort is redundant, for the reasons given above, it is the right answer. Eventually you will get used to it. However, if initialization requires access to system resources that are shared by all the parallel tasks (such as file systems and networks), having each task attempt to obtain the resources will create contention in the system and hinder the initialization process. In this case, it makes sense for one task to access the system resources on behalf of the entire application. In fact, if multiple system resources are required, you could have multiple tasks access each of the resources in parallel. Once the data has been obtained from the resource, you need to decide whether to share the raw data among the tasks and have each task process it, or have one task perform the initialization processing and distribute the results to all the other tasks. You can base this decision on whether the amount of data increases or decreases during the initialization processing. Of course, you want to transmit the smaller amount.

Duplicating the same work on all the remote tasks (which is not the same as redundancy, which implies something can be eliminated) is not bad if:
- The work is inherently serial
- The work is parallel, but the cost of computation is less than the cost of communication
- The work must be completed before tasks can proceed
- Communication can be avoided by having each task perform the same work.

Protocols supported

To perform data communication, PE interfaces with a low-level communication API (LAPI), which is a reliable transport provided with AIX. LAPI interfaces with a lower level protocol, running in the user space (User Space protocol), which offers a low-latency and high-bandwidth communication path to user applications, running over a high performance switch. LAPI alternatively interfaces with the IP layer.

| Some hardware adapters, such as InfiniBand HCA, support direct access to the
| adapter hardware, so that messages can be sent bypassing the operating system
| kernel. This mode of message passing is called 'User Space', and is supported by
| Parallel Environment on specific adapters. For optimal performance, PE uses the

User Space (US) protocol as its communication path. However, PE also lets you run parallel applications that use the IP interface of LAPI.

The User Space interface allows user applications to take full advantage of the high speed interconnect, and you should use it whenever communication is a critical issue (for instance, when running a parallel application in a production environment). With LoadLeveler, you can use the User Space interface by more than one process per node at a given time.

Both the IP and User Space interfaces allow multiple tasks per job on a single node. As a result, you can use either interface in development or test environments, where more attention is paid to the correctness of the parallel program than to its speed-up, and therefore, more users can work on the same nodes at a given time. In both cases, data exchange always occurs between processes, without involving the POE Partition Manager daemon.

Shared memory message passing

For MPI programs in which multiple tasks run on the same computing node, using shared memory to send messages between tasks may be beneficial. This applies to programs running over either the IP or User Space protocol.

By setting the **MP_SHARED_MEMORY** environment variable to *YES*, you can select the shared memory protocol. If *all* the tasks of your program run on the same node, and you specify the shared memory protocol, shared memory is used exclusively for all MPI communications.

The default for **MP_SHARED_MEMORY** is *YES*, so explicitly setting it is not required.

For more information on PE's shared memory support, see *IBM Parallel Environment: Operation and Use, Volume 1*.

To thread or not to thread - protocol implications

If you are unfamiliar with POSIX threads, do not try to learn both threads and MPI all at once. Get some experience writing and debugging single process multi-threaded programs first, then tackle multiprocess multithreaded programs.

A parallel program using MPI normally depends on task parallelism with two or more tasks (or processes) that communicate by message passing. Each of these tasks, by default, has one user thread. An application may explicitly create additional threads within each task, resulting in thread level as well as task level parallelism. If thread creation is done, the application must manage both levels of parallelism properly.

While each threaded task has more than one independent instruction stream, all of a task's threads share the same address space, file system, and environment variables. In addition, all the threads in a threaded MPI task have the same MPI communicators, data types, ranks, and so forth.

In each threaded MPI task, the **MPI_INIT** routine must be called before any thread can make an MPI call, and all MPI calls must be completed before **MPI_FINALIZE** is called. The principal difference between a threaded task and a non-threaded task is that, in each threaded task, more than one blocking call may be in progress at any given time.

The underlying communication subsystem provides thread-dispatching, so that all blocking messages are given a chance to run when a message completes.

The MPI library creates the following service threads:
- A thread that periodically wakes up and calls the message passing dispatcher, and handles interrupts generated by arriving packets.
- Responder threads used to implement MPI I/O.

The service threads above are terminated when **MPI_FINALIZE** is called. These threads are not available to end users.

Thread debugging implications

To effectively debug the application, you must be aware of how threads are dispatched. When a task is stopped, all threads are stopped. Each time you issue an execution command, such as **step over**, **step into**, **step return**, or **continue**, all the threads are released for execution until the next stop (at which time they are stopped, even if they have not completed their work). This stop may be at a breakpoint you set or the result of a step. A single step over an MPI routine may prevent the MPI library threads from completely processing the message that is being exchanged.

For example, if you wanted to debug the transfer of a message from a send node to a receiver node, you would step over an MPI_SEND in your program on task 1, switch to task 2, then step over the MPI_RECV on task 2. Unless the MPI threads on task 1 and 2 have the opportunity to process the message transfer, it will appear that the message was lost. Remember that the window of opportunity for the MPI threads to process the message is brief, and is open only during the **step over**. Otherwise, the threads will be stopped. Longer-running execution requests, of both the sending and receiving nodes, allow the message to be processed and, eventually, received.

For more information on debugging threaded and non-threaded MPI programs with the PE debugging tool, (**pdbx**), see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*, which provides more detailed information on how to manage and display threads.

For more information on the threaded MPI library, see *IBM Parallel Environment: MPI Programming Guide*.

Checkpointing and restarting a parallel program

Checkpointing a parallel program is a mechanism for temporarily saving the state of a parallel program at a specific point (*checkpointing*), and then later restarting it from the saved state. When you checkpoint a program, the checkpointing function captures the state of the application as well as all data, and saves it in a file. When the program is restarted, the restart function retrieves the application information from the file it saved. The program then starts running again from the place at which it was saved.

For more information on PE's checkpointing and restarting functions, see *IBM Parallel Environment: Operation and Use, Volume 1*.

Chapter 3. Diagnosing and correcting common problems

What do you do when something goes wrong with your parallel program? PE provides ways to identify and correct problems that arise when you are developing or executing your parallel program. This all depends on where in the process the problem occurred and what the symptoms are.

This information is probably more useful if you use it in conjunction with *IBM Parallel Environment: Operation and Use, Volume 1* and *IBM Parallel Environment for AIX: Operation and Use, Volume 2*. So, you might want to go find them, and keep them on hand for reference.

Here are the steps, greatly abbreviated, in the basic process of creating a parallel program:
1. Create and compile program
2. Start PE
3. Execute the program
4. Verify the output
5. Optimize the performance.

Problems can arise in any one of these steps, and knowing which tools to use to identify, analyze and correct the problem is the first step. The remainder of this section describes some of the common problems you might run into, and what to do when they occur. The problems in this section are labeled according to the *symptom* you might be experiencing.

Messages

Messages are an important part of diagnosing problems, so it is essential that you have access to them and that they are at the correct level.

Message catalog errors

You may get message catalog errors. This usually means that the message catalog could not be located or loaded. Check that your **NLSPATH** environment variable includes the path where the message catalog is located. The environment variable **NLSPATH** is used by the various PE components to find the appropriate message catalogs. If the message catalogs are not in the proper place, or your environment variables are not set properly, your system administrator can help.. Refer your system administrator to "National language support (NLS)" on page xii for more information.

The following are the PE message catalogs:
- pepoe.cat
- pempl.cat
- pepdbx.cat
- peperf.cat

Finding PE messages

There are a number of places that you can find PE messages:
- PE messages are displayed on the home node when it is running POE (STDERR and STDOUT).
- If you set either the **MP_PMDLOG** environment variable or the **-pmdlog** command line option to **yes**, PE messages are collected in the pmd log file of each task, in **/tmp** (STDERR and STDOUT).

You can also use LookAt to look up message explanations. For more information on how to do this see “Using LookAt to look up message explanations” on page xii

Logging POE errors to a file

You can also specify that diagnostic messages be logged to a file in **/tmp** on each of the remote nodes of your partition by using the **MP_PMDLOG** environment variable. The log file is called **/tmp/mplog.***jobid.taskid*, where *jobid* is a unique identifier and *taskid* is the task number. The *jobid* is the same for all remote nodes. This file contains additional diagnostic information about why the user connection was not made. If the file is not there, then pmd did not start. Check the **/etc/inetd.conf** and **/etc/services** entries and the executability of pmd for the root user ID again.

For more information about the **MP_PMDLOG** environment variable, see *IBM Parallel Environment: Operation and Use, Volume 1*.

Message format

Knowing which component a message is associated is helpful when trying to resolve a problem. PE messages include prefixes that identify the related component. The message identifiers for the PE components are as follows:

0029-*nnnn*	Parallel debugger (**pdbx**)
0031-*nnn*	Parallel Operating Environment
0032-*nnn*	Message Passing Interface
2554-*nnn*	PE Benchmarker
2554-9*nn*	Unified Trace Environment (UTE)
2660-*nnnn*	LAPI

where:

- The first four, five, or six digits (0029, 0031, 0032, 2537, 2554, or 2554-9) identify the PE component that issued the message.
- The last two, three, or four digits identify the sequence of the message in the group.

For more information about PE messages, see *IBM Parallel Environment: Messages*.

Diagnosing problems using IVP

The *Installation Verification Program* (IVP) can be a useful tool for diagnosing problems. When you installed POE, you verified that everything turned out correctly by running the IVP. It verified that the:

- Location of the libraries was correct
- Binaries existed
- Partition Manager daemon was executable
- POE files were in order
- Sample IVP programs compiled correctly.

The IVP can provide some important first clues when you experience a problem, so you may want to rerun this program before you do anything else.

Cannot compile a parallel program

Programs for PE must be compiled with the current release of the compiler scripts you are using, such as **mpxlf_r**, **mpcc_r**, or **mpCC_r** . If the command you are trying to use cannot be found, make sure the installation was successful and that your *PATH* environment variable contains the path to the compiler scripts. These commands call the Fortran, C, and C++ compilers respectively, so you also need to make sure that the underlying compiler is installed and accessible. Your system administrator should be able to assist you in verifying these things.

Cannot start a parallel job

Once you have successfully compiled your program, you either invoke it directly or start POE and then submit the program to it. In both cases, POE is started to establish communication with the parallel nodes. Problems that can occur at this point include: POE does not start, or cannot connect to the remote nodes.

These problems can be caused by other problems on the home node (where you are trying to submit the job), on the remote parallel nodes, or in the communication subsystem that connects them. You need to make sure that all the things POE expects to be set up really are set up. Here is what you do:

1. Make sure that you can execute POE. If you are a Korn shell user, type:

   ```
   $ whence poe
   ```

   If you are a C shell user, type:

   ```
   $ which poe
   ```

   If the result is just the shell prompt, you do not have POE in your path. It might mean that POE is not installed, or that your path does not point to it. Check that the file **/usr/lpp/ppe.poe/bin/poe** exists and is executable, and that your PATH includes the directory **/usr/lpp/ppe.poe/bin**.
2. Type:

   ```
   $ env | grep MP_
   ```

   Look at the settings of the environment variables beginning with **MP_**, (the POE environment variables). Check their values against what you expect, particularly **MP_HOSTFILE** (where the list of remote host names is to be found), **MP_RESD** (whether a job management system is to be used to allocate remote hosts) and **MP_RMPOOL** (the pool from which the job management system is to allocate remote hosts) values. If they are all not set, make sure that you have a file named **host.list** in your current directory. This file must include the names of all the remote parallel hosts that can be used. There must be at least as many hosts available as the number of parallel processes you specified with the **MP_PROCS** environment variable.
3. Type:

   ```
   $ poe -procs 1
   ```

   You should get the following message:

   ```
        0031-503   Enter program name and flags for each node: _
   ```

   If you do get this message, POE has successfully loaded and established communication with the first remote host in your host list file. It has also validated your use of that remote host, and is ready to go to work. If you type a command, for example, **date**, **hostname**, or **env**, you should get a response when the command executes on the remote host (like you would from **rsh**).

   If you get some other set of messages, then the message text should give you some idea of where to look. Some common situations include:

   - Cannot connect with the remote host

The path to the remote host is unavailable. Check to make sure that you are trying to connect to the host you think you are. If you are using LoadLeveler to allocate nodes from a pool, you may want to allocate nodes from a known list instead. **ping** the remote hosts in the list to see if a path can be established to them. If it can, run **rsh** *remote_host* **date** to verify that the remote host can be contacted and recognizes the host from which you submitted the job, so it can send results back to you.

Check the **/etc/services** file on your home node, to make sure that the Parallel Environment service is defined. Check the **/etc/services** and **/etc/inetd.conf** files on the remote host to make sure that the PE service is defined, and that the Partition Manager Daemon (**pmd**) program invoked by **inetd** on the remote node is executable.

For more information on configuring **rsh** and **inetd**, see *IBM Parallel Environment: Installation*.

- User not authorized on remote host

  You need an ID on the remote host and your ID on the home host (the one from which you are submitting the job) must be authorized to run commands on the remote hosts. You do this by placing a **$HOME/.rhosts** file on the remote hosts that identify your home host and ID. Brush up on “Access” on page 2 if you need to. Even if you have a **$HOME/.rhosts** file, make sure that you are not denied access the **/etc/hosts.equiv** file on the remote hosts.

  In some installations, your home directory is a mounted file system on both your home node and the remote host. In this case, check with your system administrator.

  Even if the remote host is actually the same machine as your home node, you still need an entry in the **.rhosts** file.

- Other strangeness

  On the home node, you can set or increase the **MP_INFOLEVEL** environment variable (or use the **-infolevel** command line option) to get more information out of POE while it is running. Although this does not give you any more information about the error, or prevent it, it gives you an idea of where POE was, and what it was trying to do when the error occurred. A value of 6 gives you more information than you could ever want. See Appendix A, “A sample program to illustrate messages,” on page 97 for an example of the output from this setting.

Cannot execute a parallel program

Once POE can be started, you need to consider the problems that can arise in running a parallel program, specifically initializing the message passing subsystem. The way to eliminate this initialization as the source of POE startup problems is to run a program that does not use message passing.

As discussed in “Running POE” on page 3, you can use POE to invoke a command or serial program on remote nodes. If you can get a command or simple program, like *Hello, World!*, to run under POE, but a parallel program does not, you can be pretty sure the problem is in the message passing subsystem. The message passing subsystem is the underlying implementation of the message passing calls used by a parallel program (in other words, an **MPI_SEND**). POE code that is linked into your executable by the compiler script (**mpcc_r**, **mpCC_r**, **mpxlf_r**) initializes the message passing subsystem.

The Parallel Operating Environment (POE) supports two distinct communication subsystems, an IP-based system, and User Space optimized adapter support. The subsystem choice is normally made at run time, by environment variables or command line options passed to POE. Use the IP subsystem for diagnosing initialization problems before worrying about the User Space (US) subsystem. Select the IP subsystem by setting the environment variable:

```
$ export MP_EUILIB=ip
```

Use specific remote hosts in your host list file and do not use LoadLeveler (set **MP_RESD=no**). If you do not have a small parallel program, compile the following sample program, **hello_parallel_world**.

Here is the **hello_parallel_world** program in C:

```
/*********************************************************************
*
* Hello Parallel World C Example
*
* To compile:
* mpcc -o hello_parallel_world_c hello_parallel_world.c
*
**********************************************************************/
#include<stdlib.h>
#include<stdio.h>
#include<mpi.h>
/* Basic program to demonstrate compilation and execution techniques */
int main()
{
MPI_Init(0,0);
printf("Hello, Parallel World!\n");
MPI_Finalize();
exit(0);
}
```

You compile it in C like this:

```
$ mpcc_r -o hello_parallel_world_c hello_parallel_world.c
```

And here is the **hello_parallel_world** program in Fortran:

```
c*********************************************************************
c*
c* Hello World Fortran Example
c*
c* To compile:
c* mpfort -o hello_parallel_world_f hello_parallel_world.f
c*
c*********************************************************************
c ------------------------------------------------------------------
c  Basic program to demonstrate compilation and execution techniques
c ------------------------------------------------------------------
c     program  hello

implicit none
include mpif.h
INTEGER error
MPI_INIT(error)
write(6,*)'Hello, Parallel World!'
MPI_FINALIZE(error)

stop
end
```

You compile it in Fortan like this:

```
$ mpxlf_r -o hello_parallel_world_f hello_parallel_world.f
```

Make sure that the executable can be loaded on the remote host that you are using.

Type the following command, and then look at the messages on the console. For C, type the command like this:

```
$ poe hello_parallel_world_c -procs 1 -infolevel 4
```

For Fortran, type the command like this:

```
$ poe hello_parallel_world_f -procs 1 -infolevel 4
```

If you get

```
Hello, Parallel World!
```

then the communication subsystem has been successfully initialized on the one node and things should be looking good. Just for kicks, make sure that there are two remote nodes in your host list file and try again with the following command. If you are using C, type the command like this:

```
$ poe hello_parallel_world_c -procs 2
```

If you are using Fortran, type the command like this:

```
$ poe hello_parallel_world_f -procs 2
```

If and when **hello_parallel_fortran** works with IP and device en0 (the Ethernet), try again with the high speed interconnect.

Each node has one name that it is known by on the external LAN to which it is connected, and another name that it is known by on the interconnect. If the node name you use is not the proper name for the network device you specify, the connection is not made. You can put the names in your host list file. Otherwise, use LoadLeveler to locate the nodes.

The following example assumes you are using C,

```
$ export MP_RESD=yes
$ export MP_EUILIB=ip
$ export MP_EUIDEVICE=sn_single
$ poe hello_parallel_world_c -procs 2 -ilevel 2
```

where **sn_single** is the switch device name. Look at the console lines containing the string **MPI euidevice**. These identify the device name that is actually being used for message passing (as opposed to the IP address that is used to connect the home node to the remote hosts.) If these are not device names, check the LoadLeveler configuration and the switch configuration.

Once IP works, and you are on a clustered server, you can try message passing using the User Space device support, if User Space is supported in your environment. Note that LoadLeveler allows you to run multiple tasks over the switch adapter while in User Space.

You can run **hello_parallel_world** with the User Space library by typing the following. This example assumes you are using C.

```
e$ export MP_RESD=yes
$ export MP_EUILIB=us
$ export MP_EUIDEVICE=sn_single
$ poe hello_parallel_world_c -procs 2 -ilevel 6
```

The console log should inform you that you are using User Space support, and that LoadLeveler is allocating the nodes for you. LoadLeveler tells you that it cannot allocate the requested nodes if someone else is already running on them **and** has requested dedicated use of the switch, or if User Space capacity has been exceeded.

You can try for other specific nodes, or you can ask LoadLeveler for nonspecific nodes from a pool. You can refer to *IBM Parallel Environment: Operation and Use, Volume 1*.

The program runs but...

Using the parallel debugger

An important tool in analyzing your parallel program is the PE parallel debugger (pdbx). In some situations, using a parallel debugger is just like using a debugger for a serial program. In other situations, however, the parallel nature of the problem introduces some subtle and not-so-subtle differences which you should understand to use the debugger efficiently. While debugging a serial application, you can focus your attention on the single problem area. In a parallel application, you have to shift your attention between the various parallel tasks and also consider how the interaction among the tasks may be affecting the problem.

The simplest problem

The simplest parallel program to debug is one where all the problems exist in a single task. In this case, you can unhook all the other tasks from the debugger's control and use the parallel debugger as if it were a serial debugger. However, this case is also the most rare.

The next simplest problem

The next simplest case is one where all the tasks are doing the same thing and they all experience the problem that is being investigated. In this case, you can apply the same debug commands to all the tasks, advance them in lockstep and interrogate the state of each task before proceeding. In this situation, you need to be sure to avoid debugging-introduced deadlocks. These are situations where the debugger is trying to single-step a task past a blocking communication call, but the debugger has not stepped the sender of the message past the point where the message is sent. In these cases, control will not be returned to the debugger until the message is received, but the message will not be sent until control returns to the debugger.

OK, the worst problem

The most difficult situation to debug, and also the most common, is where not all the tasks are doing the same thing and the problem spans two or more tasks. In these situations, you have to be aware of the state of each task, and the interrelations among tasks. You must ensure that blocking communication events either have been or will be satisfied before stepping or continuing through them. This means that the debugger has already executed the send for blocking receives, or the send will occur at the same time (as observed by the debugger) as the receive. Frequently, you may find that tracing back from an error state leads to a message from a task to which you were not paying attention. In these situations, your only choice may be to run the application again and focus on the events leading up to the send.

When a core dump is created

If your program creates a core dump, POE saves a copy of the core file so you can debug it later. Unless you specify otherwise, POE saves the core file in the **coredir**.*taskid* directory, under the current working directory, where *taskid* is the task number. For example, if your current directory is **/u/mickey**, and your application creates a core dump (segmentation fault) while running on the node that is task 4, the core file will be located in **/u/mickey/coredir.4** on that node.

You can control where POE saves the core file by using the **-coredir** POE command line option or the **MP_COREDIR** environment variable.

Standard AIX corefiles can be large and often the information in the files appears at a very low level. This can make the files difficult to debug. These large files can also consume too much disk space, CPU time, and network bandwidth. To avoid this problem, PE allows you to produce corefiles in the *Ptools Lightweight Corefile Format*. Lightweight corefiles provide simple shared stack traces (listings of function calls that led to the error), and consume less system resources than traditional corefiles. For more information on lightweight corefiles and how to generate them, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.

Debugging core dumps

There are two ways you can use traditional core dumps to find problems in your program. After running the program, you can examine the resulting core file to see if you can find the problem. Or, you can try to view your program state by *catching* it at the point where the problem occurs.

Examining core files: Before you can debug a core file, you first need to get one. Let's just generate it. The following example is an MPI program in which even-numbered tasks pass the *answer to the meaning of life* to odd-numbered tasks. It is called **bad_life.c**, and here is what it looks like:

```
/*******************************************************************
*
* bad_life program

* To compile:
* mpcc -g -o bad_life bad_life.c
*
*******************************************************************/

#include <stdio.h>
#include <mpi.h>

void main(int argc, char *argv[])
{
        int  taskid;
        MPI_Status  stat;

        /* Find out number of tasks/nodes. */
        MPI_Init( &argc, &argv);
        MPI_Comm_rank( MPI_COMM_WORLD, &taskid);

        if ( (taskid % 2) == 0)
        {
                char *send_message = NULL;

                send_message = (char *) malloc(10);
                strcpy(send_message, "Forty Two");
                MPI_Send(send_message, 10, MPI_CHAR, taskid+1, 0,
                        MPI_COMM_WORLD);
                free(send_message);
        } else
        {
                char *recv_message = NULL;

                MPI_Recv(recv_message, 10, MPI_CHAR, taskid-1, 0,
                MPI_COMM_WORLD, &stat);
                printf("The answer is  %s\n", recv_message);
                free(recv_message);
        }
                printf("Task %d complete.\n",taskid);
                MPI_Finalize();
                exit(0);
}
```

bad_life.c was compiled with the following parameters:

```
$ mpcc -g bad_life.c -o bad_life
```

and when it runs, you get the following results:

```
$ export MP_PROCS=4
$ export MP_LABELIO=yes
$ bad_life
  0:Task 0 complete.
  2:Task 2 complete.
ERROR: 0031-250  task 1: Segmentation fault
ERROR: 0031-250  task 3: Segmentation fault
ERROR: 0031-250  task 0: Terminated
ERROR: 0031-250  task 2: Terminated
```

As you can see, **bad_life.c** gets two segmentation faults which generate two core files. If you list the current directory, you can see two core files; one for task 1 and the other for task 3.

```
$ ls -lR core*
total 88
-rwxr-xr-x   1 hoov     staff       8472 May 02 09:14 bad_life
-rw-r--r--   1 hoov     staff        928 May 02 09:13 bad_life.c
drwxr-xr-x   2 hoov     staff        512 May 02 09:01 coredir.1
drwxr-xr-x   2 hoov     staff        512 May 02 09:36 coredir.3
-rwxr-xr-x   1 hoov     staff       8400 May 02 09:14 good_life
-rw-r--r--   1 hoov     staff        912 May 02 09:13 good_life.c
-rw-r--r--   1 hoov     staff         72 May 02 08:57 host.list
./coredir.1:
total 48
-rw-r--r--   1 hoov     staff      24427 May 02 09:36 core

./coredir.3:
total 48
-rw-r--r--   1 hoov     staff      24427 May 02 09:36 core
```

Run **dbx** on one of the core files to find the problem. You run **dbx** like this:

```
$ dbx bad_life coredir.1/core

Type 'help' for help.
[using memory image in coredir.1/core]
reading symbolic information ...

Segmentation fault in . at 0xf014
0x0000f014 warning: Unable to access address 0xf014 from core
```

Now, let's see where the program crashed and what its state was at that time. If you issue the **where** command,

```
(dbx) where
```

You can see the program stack:

```
warning: Unable to access address 0xf014 from core
warning: Unable to access address 0xf014 from core
warning: Unable to access address 0xf010 from core
warning: Unable to access address 0xf010 from core
warning: Unable to access address 0xf014 from core
warning: Unable to access address 0xf014 from core
warning: Unable to access address 0xf010 from core
warning: Unable to access address 0xf010 from core
warning: Unable to access address 0xf014 from core
.() at 0xf014
lapi_recv_vec(??, ??, ??, ??, ??, ??) at 0xd2ccc298
process_hdr_hndlr_contig(0x0, 0x0, 0xf15433d8, 0x202b5368, 0x20a4bb88) at 0xd31d
58c0
_lapi_recv_callback(0x0, 0x20a4bb88, 0x2000) at 0xd31d6a10
udp_read_dgsp(0x0, 0xf1542608, 0x0, 0x0) at 0xd05b9294
_receive_processing(0x0) at 0xd31d41b0
_lapi_dispatcher(0x0, 0x0) at 0xd3193cf4
_lapi_msgpoll_internal(0x0, 0x1, 0x2ff225e8, 0x0, 0x0) at 0xd31bb8f0
LAPI_Msgpoll(0x0, 0x1, 0x2ff225e8) at 0xd31bfc60
mpci_recv(??, ??, ??, ??, ??, ??, ??, ??) at 0xd2cc6ce0
_mpi_recv(??, ??, ??, ??, ??, ??, ??) at 0xd3078e94
MPI__Recv(??, ??, ??, ??, ??, ??, ??) at 0xd3075d44
unnamed block $b2, line 36 in "bad_life.c"
main(argc = 1, argv = 0x2ff229bc), line 36 in "bad_life.c"
(dbx)
```

The output of the **where** command shows that **bad_life.c** failed at line 36, like this:

```
(dbx) func main
(dbx) list 36
   36                 MPI_Recv(recv_message, 10, MPI_CHAR, taskid-1, 0,
```

Look at line 36 of **bad_life.c**. The first guess is that one of the parameters being passed into **MPI_RECV** is bad. Look at some of these parameters to see if you can find the source of the error. For example:

```
(dbx) print recv_message
"recv_message" is not active
```

| The receive buffer pointer has been initialized to NULL rather than the address of a
| valid buffer. The sample programs include a solution called **good_life.c**.

Compiling **bad_life.c** with the **-g** compile flag gives all the debugging information you need to view the entire program state and to print program variables. If you did not compile the program with the **-g** flag, and if you turned optimization on (**-O**), there is virtually no information to tell you what happened when the program executed. If this is the case, you can still use **dbx** to look at only stack information, which allows you to determine the function or subroutine that generated the core dump.

Viewing the program state: If collecting core files is impractical, you can also try *catching* the program at the segmentation fault. You do this by running the program under the control of the debugger. The debugger gets control of the application at the point of the segmentation fault, and this allows you to view your program state at the point where the problem occurs.

The following example uses **bad_life** again, but uses **pdbx** instead of **dbx**. Load **bad_life** under **pdbx** with the following command:

```
> pdbx bad_life -procs 4 -hfile /u/voe3/>
pdbx Version 4, Release 1.1  -- Feb  5 2004 18:31:06

   0:Core file "
   0:" is not a valid core file (ignored)
   2:Core file "
   2:" is not a valid core file (ignored)
   1:Core file "
   1:" is not a valid core file (ignored)
   3:Core file "
   3:" is not a valid core file (ignored)
   0:reading symbolic information ...
   1:reading symbolic information ...
   1:[1] stopped in main at line 20 ($t1)
   1:   20      MPI_Init( &argc, &argv);
   3:reading symbolic information ...
   2:reading symbolic information ...
   0:[1] stopped in main at line 20 ($t1)
   0:   20      MPI_Init( &argc, &argv);
   3:[1] stopped in main at line 20 ($t1)
   3:   20      MPI_Init( &argc, &argv);
   2:[1] stopped in main at line 20 ($t1)
   2:   20      MPI_Init( &argc, &argv);
0031-504  Partition loaded ...
```

Next, let the program run to allow it to reach a segmentation fault.

```
pdbx(all) cont
   0:Task 0 complete.
   2:Task 2 complete.
   1:
   1:Segmentation fault in . at 0xf014 ($t1)
   1:0x0000f014 7ca01d2a      stswx   r5,r0,r3
   3:
   3:Segmentation fault in . at 0xf014 ($t1)
   3:0x0000f014 7ca01d2a      stswx   r5,r0,r3
```

Once you get segmentation faults, you can focus your attention on one of the tasks that failed. Look at task 1:

```
pdbx(all) on 1
```

By using the **pdbx where** command, you can see where the problem originated in the source code:

```
pdbx(1) where
   1:.() at 0xf014
   1:lapi_recv_vec(??, ??, ??, ??, ??, ??) at 0xd2ccc298
   1:process_hdr_hndlr_contig(0x0, 0x0, 0xf15433d8, 0x202b5368, 0x20a35b88) at
       0xd31d58c0
   1:_lapi_recv_callback(0x0, 0x20a35b88, 0x2000) at 0xd31d6a10
   1:udp_read_dgsp(0x0, 0xf1542608, 0x0, 0x0) at 0xd0aa3294
   1:_receive_processing(0x0) at 0xd31d41b0
   1:_lapi_dispatcher(0x0, 0x0) at 0xd3193cf4
   1:_lapi_msgpoll_internal(0x0, 0x3e8, 0x2ff225b8, 0x0, 0x0) at 0xd31bb8f0
   1:LAPI_Msgpoll(0x0, 0x186a0, 0x2ff225b8) at 0xd31bfc60
   1:mpci_recv(??, ??, ??, ??, ??, ??, ??, ??) at 0xd2cc7048
   1:_mpi_recv(??, ??, ??, ??, ??, ??, ??) at 0xd3078e94
   1:MPI__Recv(??, ??, ??, ??, ??, ??, ??) at 0xd3075d44
   1:unnamed block $b2, line 36 in "bad_life.c"
   1:main(argc = 1, argv = 0x2ff2298c), line 36 in "bad_life.c"
```

Now, let's move up the stack to **function main**:

```
pdbx(1) func main
```

Next, list line 36, which is where the problem is located:

```
pdbx(1) l 36
   1:   36            MPI_Recv(recv_message, 10, MPI_CHAR, taskid-1, 0,
```

Print the value of **recv_message**:

```
pdbx(1) p recv_message
   1:"recv_message" is not active
```

The program passes a bad parameter to **MPI_RECV**.

Both the techniques help you find the location of the problem in your code. The example used makes it look easy, but in many cases it will not be so simple. However, knowing where the problem occurred is valuable information if you are forced to debug the problem interactively.

On the lighter side...: One of the new features in POE is the ability to capture more detailed information about a program when it abnormally terminates, while also reducing the amount of space needed for it. POE has the ability to produce Light Weight Core Files, as opposed to standard AIX core files. This greatly reduces the size of the core files while greatly enhancing the information that is produced.

First, you need to tell POE to produce Light Weight Core Files, with the **-corefile_format** flag or **MP_COREFILE_FORMAT** environment variable.

```
> bad_life -procs 4 -labelio yes -corefile_format lwcf              <
   0:Task 0 complete.
   2:Task 2 complete.
ERROR: 0031-250  task 1: Segmentation fault
ERROR: 0031-250  task 0: Terminated
ERROR: 0031-250  task 2: Terminated
ERROR: 0031-250  task 3: Segmentation fault
```

You will notice the same program output, however, now when you look in the **coredir.1** and **coredir.3** directories, you begin to see the difference.

```
> cd coredir.1
```

Now look in the directory.

```
> ls -lt
total 30768
-rw-r--r--   1 voe3     usr           1269 Feb 19 13:40 lwcf
-rw-r--r--   1 voe3     usr       15745755 Feb 19 13:16 core
```

You should notice two differences. First, there is a second file, named **lwcf** (or whatever the file name specified by the **-corefile_format** option or **MP_COREFILE_FORMAT** environment variable), in addition to the file named **core**. The second difference is in the file sizes - the standard AIX core files are much larger. Now look at what you have in the new file.

The new file is a text output file, that can be viewed with any text viewer or **vi**. It will contain output produced by the Light Weight Core File facility, containing stack and thread traces for the entire program. To keep it simple, use **cat** to view the file:

```
>cat lwcf
+++PARALLEL TOOLS CONSORTIUM LIGHTWEIGHT COREFILE FORMAT version 1.1
+++LCB 1.0 Wed Feb 19 13:39:14 2004 Generated by IBM AIX 5.3
#
+++ID Node 1 Process 622736 Thread 1
***FAULT "SIGSEGV - Segmentation violation"
+++STACK
# At location 0x0000f014 but procedure information unavailable.
lapi_recv_vec : 0x00000550
process_hdr_hndlr_contig : 0x00000274
```

```
_lapi_recv_callback : 0x000003c4
udp_read_dgsp : 0x000000a0
_receive_processing : 0x00000058
_lapi_dispatcher : 0x00000150
_lapi_msgpoll_internal : 0x000004a4
LAPI_Msgpoll : 0x000001ac
mpci_recv : 0x00000f38
_mpi_recv : 0x0000015c
MPI__Recv : 0x00000630
main : 36 # in file <bad_life.c>
---STACK
---ID Node 1 Process 622736 Thread 1
#
+++ID Node 1 Process 622736 Thread 2
+++STACK
sigwait : 0x000002d0
pm_async_thread : 0x000006e8
_pthread_body : 0x000000e8
---STACK
---ID Node 1 Process 622736 Thread 2
#
+++ID Node 1 Process 622736 Thread 3
+++STACK
_intr_hndlr : 0x00000228
_pthread_body : 0x000000e8
---STACK
---ID Node 1 Process 622736 Thread 3
#
+++ID Node 1 Process 622736 Thread 4
+++STACK
_event_wait : 0x0000005c
_cond_wait_local : 0x0000034c
_cond_wait : 0x00000050
pthread_cond_wait : 0x000001d8
_compl_hndlr_thr : 0x00000174
_pthread_body : 0x000000e8
---STACK
---ID Node 1 Process 622736 Thread 4
---LCB
```

The output contains a lot of information for such a small file, a true case where less is more. You can see where all of the threads were, and immediately know what caused the problem and where it is.

For completeness, switch over to the **coredir.3** directory, to see what happened with the other task that terminated abnormally.

```
> cd ../coredir.3
> ls -lt
total 30768
-rw-r--r--   1 voe3     usr           1269 Feb 19 13:40 lwcf
-rw-r--r--   1 voe3     usr       15745915 Feb 19 13:16 core
```

Here you see the same thing, two files, one large standard AIX core file, and a small Light Weight Core File. If you look at the **lwcf** file again, you will see pretty much the same thing as before, except it will show things from task 3's point of view:

```
> cat lwcf
+++PARALLEL TOOLS CONSORTIUM LIGHTWEIGHT COREFILE FORMAT version 1.1
+++LCB 1.0 Wed Feb 19 13:39:14 2004 Generated by IBM AIX 5.3
#
+++ID Node 3 Process 442600 Thread 1
***FAULT "SIGSEGV - Segmentation violation"
+++STACK
# At location 0x0000f014 but procedure information unavailable.
```

```
lapi_recv_vec : 0x00000550
process_hdr_hndlr_contig : 0x00000274
_lapi_recv_callback : 0x000003c4
udp_read_dgsp : 0x000000a0
_receive_processing : 0x00000058
_lapi_dispatcher : 0x00000150
_lapi_msgpoll_internal : 0x000004a4
LAPI_Msgpoll : 0x000001ac
mpci_recv : 0x00000f38
_mpi_recv : 0x0000015c
MPI__Recv : 0x00000630
main : 36 # in file <bad_life.c>
---STACK
---ID Node 3 Process 442600 Thread 1
#
+++ID Node 3 Process 442600 Thread 2
+++STACK
sigwait : 0x000002d0
pm_async_thread : 0x000006e8
_pthread_body : 0x000000e8
---STACK
---ID Node 3 Process 442600 Thread 2
#
+++ID Node 3 Process 442600 Thread 3
+++STACK
_intr_hndlr : 0x00000228
_pthread_body : 0x000000e8
---STACK
---ID Node 3 Process 442600 Thread 3
#
+++ID Node 3 Process 442600 Thread 4
+++STACK
_event_wait : 0x0000005c
_cond_wait_local : 0x0000034c
_cond_wait : 0x00000050
pthread_cond_wait : 0x000001d8
_compl_hndlr_thr : 0x00000174
_pthread_body : 0x000000e8
---STACK
---ID Node 3 Process 442600 Thread 4
---LCB
```

The Light Weight Core File option gives you a quick and efficient way of seeing where things went bad, while saving some space along the way.

Core dumps and threaded programs: If a task of a threaded program produces a core file, the partial dump produced by default does not contain the stack and status information for all threads. Therefore, it has limited usefulness. You can request AIX to produce a full core file, but such files are generally larger than permitted by user limits (the communication subsystem alone generates more than 64 MB of core information). As a result, you consider two alternatives:

- Request that AIX generate a *lightweight corefile*. Lightweight corefiles contain less detail than standard AIX corefiles and, therefore, consume less disk space, CPU time, and network bandwidth. For more information about lightweight corefiles, see *IBM Parallel Environment: Operation and Use, Volume 1*.
- Use the attach capability of **dbx**, **xldb**, or **pdbx** to examine the task while it is still running.

No output at all

Should there be output?

If you are not getting output from your program and you think you ought to be, make sure you have enabled the program to send data back to you. If the **MP_STDOUTMODE** environment variable is set to a number, it is the number of the only task for which standard output will be displayed. If that task does not generate standard output, you will not see any.

There should be output

If **MP_STDOUTMODE** is set appropriately, the next step is to verify that the program is actually doing something. Start by observing how the program terminates (or fails to terminate). It will do one of the following things:

- Terminate without generating output other than POE messages.
- Fail to terminate after a **really** long time, still without generating output.

In the first case, you should examine any messages you receive. Since your program is not generating any output, all of the messages will be coming from POE.

In the second case, you will have to stop the program yourself (**<Ctrl-c>** should work).

One possible reason for lack of output could be that your program is terminating abnormally before it can generate any. POE will report abnormal termination conditions such as being killed, as well as non-zero return codes. Sometimes these messages are obscured in the blur of other errata, so it is important to check the messages carefully.

Figuring out return codes: It is important to understand POE's interpretation of return codes. If the exit code for a task is zero(0) or in the range of 2 to 127, then POE will make that task wait until all tasks have exited. If the exit code is 1 or greater than 128 (or less than 0), then POE will terminate the entire parallel job abruptly (with a **SIGTERM** signal to each task). In normal program execution, one would expect to have each program go through **exit**(0) or **STOP**, and exit with an exit code of 0. However, if a task encounters an error condition (for example, a full file system), then it may exit unexpectedly. In these cases, the exit code is usually set to -1. If, however, you have written error handlers which produce exit codes other than 1 or -1, then POE's termination algorithm may cause your program to *hang* because one task has terminated abnormally, while the other tasks continue processing (expecting the terminated task to participate).

If the POE messages indicate the job was killed (either because of some external situation like low page space or because of POE's interpretation of the return codes), it may be enough information to fix the problem. Otherwise, you may have to do more analysis.

The program hangs

If you have gotten this far and the POE messages, and the additional checking by the message passing routines, have not shed any light on why your program is not generating output, the next step is to figure out whether your program is doing anything at all (besides not giving you output).

Let's look at the following example...it has a bug in it.

```
/**********************************************************************
*
* Ray trace program with bug
*
* To compile:
* mpcc -g -o rtrace_bug rtrace_bug.c
*
*
* Description:
* This is a sample program that partitions N tasks into
* two groups, a collect node and N - 1 compute nodes.
* The responsibility of the collect node is to collect the data
* generated by the compute nodes. The compute nodes send the
* results of their work to the collect node for collection.
*
* There is a bug in this code.  Please do not fix it in this file!
*
**********************************************************************/

#include <mpi.h>

#define PIXEL_WIDTH 50
#define PIXEL_HEIGHT 50

int First_Line = 0;
int Last_Line  = 0;

void main(int argc, char *argv[])
{
  int numtask;
  int taskid;

  /* Find out number of tasks/nodes. */
  MPI_Init( &argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &numtask);
  MPI_Comm_rank( MPI_COMM_WORLD, &taskid);

  /* Task 0 is the coordinator and collects the processed pixels */
  /* All the other tasks process the pixels                      */
  if ( taskid == 0 )
    collect_pixels(taskid, numtask);
  else
    compute_pixels(taskid, numtask);

  printf("Task %d waiting to complete.\n", taskid);
  /* Wait for everybody to complete */
  MPI_Barrier(MPI_COMM_WORLD);
  printf("Task %d complete.\n",taskid);
  MPI_Finalize();
  exit();
}

/* In a real implementation, this routine would process the pixel */
/* in some manner and send back the processed pixel along with its*/
/* location.  Since you did process the pixel. all you do is      */
/* send back the location                                         */
compute_pixels(int taskid, int numtask)
{
  int  section;
  int  row, col;
  int  pixel_data[2];
  MPI_Status stat;

  printf("Compute #%d: checking in\n", taskid);

  section = PIXEL_HEIGHT / (numtask -1);
  First_Line = (taskid - 1) * section;
  Last_Line  = taskid * section;

  for (row = First_Line; row < Last_Line; row ++)
    for ( col = 0; col < PIXEL_WIDTH; col ++)
      {
         pixel_data[0] = row;
         pixel_data[1] = col;
         MPI_Send(pixel_data, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
      }
  printf("Compute #%d: done sending. ", taskid);
  return;
}

/* This routine collects the pixels.  In a real implementation, */
/* after receiving the pixel data, the routine would look at the*/
/* location information that came back with the pixel and move  */
/* the pixel into the appropriate place in the working buffer   */
/* Since you aren't doing anything with the pixel data, you don't */
/* bother and each message overwrites the previous one          */
collect_pixels(int taskid, int numtask)
{
  int  pixel_data[2];
  MPI_Status stat;
  int      mx = PIXEL_HEIGHT * PIXEL_WIDTH;

  printf("Control #%d: No. of nodes used is %d\n", taskid,numtask);
  printf("Control: expect to receive %d messages\n", mx);

  while (mx > 0)
    {
      MPI_Recv(pixel_data, 2, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
      mx--;
    }
  printf("Control node #%d: done receiving. ",taskid);
  return;
}
```

This example is from a ray tracing program that distributed a display buffer out to server nodes. The intent is that each task, other than Task 0, takes an equal number of full rows of the display buffer, processes the pixels in those rows, and then sends the updated pixel values back to the client. In the real application, the task would compute the new pixel value and send it as well, but in this example, you are just sending the row and column of the pixel. Because the client is getting the row and column location of each pixel in the message, it does not care which server each pixel comes from. The client is Task 0, and the servers are all the other tasks in the parallel job.

This example has a functional bug in it. With a little bit of analysis, the bug is probably easy to spot, and you may be tempted to fix it right away. PLEASE DO NOT!

When you run this program, you get the output shown below. Notice that the **-g** option is used when you compile the example. You are cheating a little because you know that there is going to be a problem, so you are compiling with debug information that is turned on right away.

```
$ mpcc -g -o rtrace_bug rtrace_bug.c
$ rtrace_bug -procs 4 -labelio yes
  1:Compute #1: checking in
  0:Control #0: No. of nodes used is 4
  1:Compute #1: done sending. Task 1 waiting to complete.
  2:Compute #2: checking in
  3:Compute #3: checking in
  0:Control: expect to receive 2500 messages
  2:Compute #2: done sending. Task 2 waiting to complete.
  3:Compute #3: done sending. Task 3 waiting to complete.
^C
ERROR: 0031-250  task 1: Interrupt
ERROR: 0031-250  task 2: Interrupt
ERROR: 0031-250  task 3: Interrupt
ERROR: 0031-250  task 0: Interrupt
```

No matter how long you wait, the program will not terminate until you press <**Ctrl-c**>.

So, you suspect the program is hanging somewhere. You know it starts executing because you get some messages from it. It could be a logical hang or it could be a communication hang.

Hangs and threaded programs

Coordinating the threads in a task requires careful locking and signaling. Deadlocks that occur because the program is waiting on locks that have not been released are common, in addition to the deadlock possibilities that arise from improper use of the MPI message passing calls.

Attach the debugger

Now that you have come to the conclusion that the program is hanging, use the debugger to find out why. The best way to diagnose this problem is to attach the debugger directly to the POE job.

Start up POE and run **rtrace_bug**:

```
$ rtrace_bug -procs 4 -labelio yes
```

To attach the debugger, you first need to get the process ID (PID) of the POE job, uusing the AIX **ps** command:

```
> ps -ef | grep poe
    voe3 680044 344226   0 09:52:33  pts/1  0:00 poe
```

Next, you need to start the **pdbx** debugger in attach mode by using the **-a** flag and the process ID (PID) of the POE job:

```
$ pdbx -a 680044
```

After starting the debugger in attach mode, a **pdbx** Attach screen appears.

```
> pdbx -a 680044
pdbx Version 4, Release 1.1 -- Feb  5 2004 18:31:06


To begin debugging in attach mode, select a task or tasks to attach.

Task      IP Addr             Node                        PID        Program
0      89.117.133.62      k133rp03.kgn.ibm.com            692328     rtrace_bug
1      89.117.133.62      k133rp03.kgn.ibm.com            553010     rtrace_bug
2      89.117.133.62      k133rp03.kgn.ibm.com            684222     rtrace_bug
3      89.117.133.62      k133rp03.kgn.ibm.com            594022     rtrace_bug
```

At the **pdbx** prompt enter the **attach** command followed by a list of tasks or **all**. For example, **attach 2 4 5-7** or **attach all**. You may also type **help** for more information or **quit** to exit the debugger without attaching.

The **pdbx** Attach screen contains a list of tasks from which you can choose, and for each task, the following information:

- Task — the task number
- IP — the IP address of the node on which the task or application is running
- Node — the name of the node on which the task or application is running
- PID — the process identifier of the task or application
- Program — the name of the application and arguments, if any

The paging tool used to display the menu will default to **pg –e** unless the PAGER environment variable specifies another pager. the debugger displays a list of task numbers that comprise the parallel job. The debugger obtains this information by reading a configuration file created by POE when it begins a job step.

After initiating attach mode, select the tasks to which you want to attach. Since you do not know which task or set of tasks is causing the problem, attach to all of the tasks by typing **attach all**:

```
pdbx(none) attach all
   0:Waiting to attach to process 692328 ...
   0:Successfully attached to rtrace_bug.
   1:Waiting to attach to process 553010 ...
   1:Successfully attached to rtrace_bug.
   2:Waiting to attach to process 684222 ...
   2:Successfully attached to rtrace_bug.
   3:Waiting to attach to process 594022 ...
   3:Successfully attached to rtrace_bug.
   0:reading symbolic information ...
   0:stopped in _event_sleep at 0xd00575d0 ($t2)
   0:0xd00575d0 (_event_sleep+0xa8) 80410014        lwz   r2,0x14(r1)
   1:reading symbolic information ...
   1:stopped in _event_sleep at 0xd00575d0 ($t2)
   1:0xd00575d0 (_event_sleep+0xa8) 80410014        lwz   r2,0x14(r1)
   3:reading symbolic information ...
   3:stopped in _event_sleep at 0xd00575d0 ($t2)
   3:0xd00575d0 (_event_sleep+0xa8) 80410014        lwz   r2,0x14(r1)
   2:reading symbolic information ...
   2:stopped in _event_sleep at 0xd00575d0 ($t2)
   2:0xd00575d0 (_event_sleep+0xa8) 80410014        lwz   r2,0x14(r1)
0029-2013 Debugger attached and ready.
```

The debugger attaches to the specified tasks. The selected executables are stopped wherever their program counters happen to be, and are then under the control of the debugger. **pdbx** displays information about the attached tasks using the task numbering of the original POE application partition.

Let's start by taking a look at task 0. First, change the current context to task 0 by typing **0**. Even though the program is not actually threaded, it is using threads created by the MPI library. To see the threads that are active, use the **threads** command:

```
pdbx(attached) on 0

pdbx(0) threads
   0: thread  state-k         wchan    state-u    k-tid   mode held scope function
   0: $t1     run                      running  2359441     k   no   sys  $PTRGL
   0:>$t2     run                      blocked  3301487     k   no   sys  _event_sleep
   0: $t3     wait                     running  2805923     k   no   sys  select
   0: $t4     wait  0xf10000879001d940 blocked  1937553     k   no   sys  _event_sleep
   0: $t5     zomb                   terminated 3506425     k   no   sys  pthread_exit
```

An aspect to be aware of when attempting to debug a program using threads is that when a program is stopped, it can be stopped in any of the running threads. In this example, by looking at the list of threads, the current thread you stopped in is shown with the **>** sign next to it (in this case, it is thread 2). Knowing that the program is single threaded, you need to switch to the current thread in the program, which is thread 1, using the **thread current 1** command:

```
pdbx(0) thread current 1
   0:warning: Thread is in kernel mode, not all registers can be accessed.
```

To see where you are in task 0, type **where**:

```
pdbx(0) where
   0:@ptrgl.$PTRGL() at 0xd01d0f88
   0:@raise.nsleep(??, ??) at 0xd01dedfc
   0:@raise.nsleep(??, ??) at 0xd01dedfc
   0:usleep(??) at 0xd01dea48
   0:mpci_recv_gen(??, ??, ??, ??, ??, ??, ??, ??) at 0xd0a8bb90
   0:mpci_recv(??, ??, ??, ??, ??, ??, ??, ??) at 0xd0a7c6cc
   0:_mpi_recv(??, ??, ??, ??, ??, ??, ??) at 0xd225de94
   0:MPI__Recv(??, ??, ??, ??, ??, ??, ??) at 0xd225ad44
   0:collect_pixels(taskid = 0, numtask = 4), line 101 in "rtrace_bug.c"
   0:main(argc = 1, argv = 0x2ff229bc), line 43 in "rtrace_bug.c"
```

Since the code is hung in low level routines, take a look at the highest line in the stack trace that has a line number and a file name associated with it. This indicates that source code association is available. In this case, it is the line that contains **collect_pixels**, which is 8 lines up from the entry containing **read**. To look more closely at the **collect_pixels** routine, type **up 8**:

```
pdbx(0) up 8
   0:collect_pixels(taskid = 0, numtask = 4), line 101 in "rtrace_bug.c"
```

Now, you can list the source code starting at the calling routine in **collect_pixels**:

```
pdbx(0) list
   0:  101        MPI_Recv(pixel_data, 2, MPI_INT, MPI_ANY_SOURCE,
   0:  102            MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
   0:  103        mx--;
   0:  104      }
   0:  105    printf("Control node #%d: done receiving. ",taskid);
   0:  106    return;
   0:  107  }
   0:  108
```

Now you can see that task 0 is stopped on a **MPI_RECV** call. To look at the local data values, type **dump**:

```
pdbx(0) dump
   0:collect_pixels(taskid = 0, numtask = 4), line 101 in "rtrace_bug.c"
   0:stat = (source = 2, tag = 0, error = -804052736, val1 = 8, val2 = 0, val3 = 800,
 val4 = 2, val5 = -559038737)
   0:mx = 100
   0:__func__ = "collect_pixels"
   0:pixel_data = (31, 49)
```

When you look at the Local Data Values, you find that variable **mx** is still set to 100, so task 0 thinks it is still going to receive 100 messages. Now take a look at what the other messages are doing. To get the stack information on task 1, switch to that task (subcommand **on 1**), then go the current running thread (thread 1, subcommand **thread current 1**):

```
pdbx(0) on 1

pdbx(1) thread current 1
   1:warning: Thread is in kernel mode, not all registers can be accessed.
pdbx(1) where
   1:@ptrgl.$PTRGL() at 0xd01d0f88
   1:@raise.nsleep(??, ??) at 0xd01dedfc
   1:@raise.nsleep(??, ??) at 0xd01dedfc
   1:usleep(??) at 0xd01dea48
   1:mpci_recv(??, ??, ??, ??, ??, ??, ??, ??) at 0xd0a7c4c4
   1:barrier_shft_b(??) at 0xd2270438
   1:_mpi_barrier(??, ??, ??) at 0xd226fb7c
   1:MPI__Barrier(??) at 0xd226e678
   1:main(argc = 1, argv = 0x2ff229b4), line 49 in "rtrace_bug.c"
```

Task 1 has reached an **MPI_BARRIER** call. If you quickly check the other tasks, you see that they have all reached this point as well.

```
pdbx(1) on 2

pdbx(2) thread current 1
   2:warning: Thread is in kernel mode, not all registers can be accessed.

pdbx(2) where
   2:@ptrgl.$PTRGL() at 0xd01d0f88
   2:@raise.nsleep(??, ??) at 0xd01dedfc
   2:@raise.nsleep(??, ??) at 0xd01dedfc
   2:usleep(??) at 0xd01dea48
   2:mpci_recv(??, ??, ??, ??, ??, ??, ??, ??) at 0xd0a7c4c4
   2:barrier_shft_b(??) at 0xd2270438
   2:_mpi_barrier(??, ??, ??) at 0xd226fb7c
   2:MPI__Barrier(??) at 0xd226e678
   2:main(argc = 1, argv = 0x2ff229b4), line 49 in "rtrace_bug.c"

pdbx(2) on 3

pdbx(3) thread current 1

pdbx(3) where
   3:_p_nsleep(??, ??) at 0xd005b7f4
   3:@raise.nsleep(??, ??) at 0xd01dedfc
   3:usleep(??) at 0xd01dea48
   3:mpci_recv(??, ??, ??, ??, ??, ??, ??, ??) at 0xd0a7c4c4
   3:barrier_shft_b(??) at 0xd2270438
   3:_mpi_barrier(??, ??, ??) at 0xd226fb7c
   3:MPI__Barrier(??) at 0xd226e678
   3:main(argc = 1, argv = 0x2ff229b4), line 49 in "rtrace_bug.c"
```

Problem solved. Tasks 1 through 3 have completed sending messages, but task 0 still expects to receive more. Task 0 was expecting 2500 messages but only received 2400, so it is still waiting for 100 messages. To see how many messages each of the other tasks are sending, look at the global variables **First_Line** and **Last_Line**.

You can get the values of **First_Line** and **Last_Line** for all of the tasks by first changing the context to attached by issuing subcommand **on attached** and then issuing subcommand **print**:

```
pdbx(1) on attached

pdbx(attached) thread current 1
   0:warning: Thread is in kernel mode, not all registers can be accessed.
   1:warning: Thread is in kernel mode, not all registers can be accessed.
   2:warning: Thread is in kernel mode, not all registers can be accessed.
pdbx(attached) print First_Line
   0:0
   1:0
   2:16
   3:32

pdbx(attached) print Last_Line
```

```
  0:0
  1:16
  2:32
  3:48
```

As you can see:

- Task 1 is processing lines 0 through 16
- Task 2 is processing lines 16 through 32
- Task 3 is processing lines 32 through 48

So, what happened to lines 48 and 49? Since each row is 50 pixels wide, and you are missing 2 rows, that explains the 100 missing messages. The division of the total number of lines by the number of tasks is not integral, so you lose part of the result when it is converted back to an integer. Where each task is supposed to be processing 16 and two-thirds lines, it is only handling 16.

Fix the problem

To fix this problem permanently, you can proceed in one of the following ways:

- Have the last task always go to the last row as you did in the debugger.
- Have the program refuse to run unless the number of tasks are evenly divisible by the number of pixels (a rather harsh solution).
- Have tasks process the complete row when they have responsibility for half or more of a row.

Since Task 1 was responsible for 16 and two thirds rows, it would process rows 0 through 16. Task 2 would process rows 17 through 33, and Task 3 would process rows 34 through 49. The way to solve it is by creating blocks, with as many rows as there are servers. Each server is responsible for one row in each block (the offset of the row in the block is determined by the server's task number). The fixed code is shown in the following example. Note that this is only part of the program.

```
/************************************************************************
*
* Ray trace program with bug corrected
*
* To compile:
* mpcc -g -o rtrace_good rtrace_good.c
*
*
* Description:
* This is part of a sample program that partitions N tasks into
* two groups, a collect node and N - 1 compute nodes.
* The responsibility of the collect node is to collect the data
* generated by the compute nodes. The compute nodes send the
* results of their work to the collect node for collection.
*
* The bug in the original code was due to the fact that each processing
* task determined the rows to cover by dividing the total number of
* rows by the number of processing tasks.  If that division was not
* integral, the number of pixels processed was less than the number of
* pixels expected by the collection task and that task waited
* indefinitely for more input.
*
* The solution is to allocate the pixels among the processing tasks
* in such a manner as to ensure that all pixels are processed.
*
************************************************************************/

compute_pixels(int taskid, int numtask)
{
  int  offset;
  int  row, col;
  int  pixel_data[2];
  MPI_Status stat;

  printf("Compute #%d: checking in\n", taskid);

  First_Line = (taskid - 1);
     /* First n-1 rows are assigned */
     /* to processing tasks         */
  offset = numtask - 1;
     /* Each task skips over rows   */
     /* processed by other tasks    */

     /* Go through entire pixel buffer, jumping ahead by numtask-1 each time */
for (row = First_Line; row < PIXEL_HEIGHT; row += offset)
  for ( col = 0; col < PIXEL_WIDTH; col ++)
    {
      pixel_data[0] = row;
      pixel_data[1] = col;
      MPI_Send(pixel_data, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
  printf("Compute #%d: done sending. ", taskid);
  return;
}
```

This program is the same as the original one except for the loop in **compute_pixels**. Now, each task starts at a row determined by its task number and jumps to the next block on each iteration of the loop. The loop is terminated when the task jumps past the last row (which will be at different points when the number of rows is not evenly divisible by the number of servers).

Why did the program hang?

The symptom of the problem in the **rtrace_bug** program was a hang. Hangs can occur for the same reasons they occur in serial programs (in other words, loops without exit conditions). They may also occur because of message passing deadlocks or because of some subtle differences between the parallel and sequential environments.

Using the debugger to analyze sometimes indicates that the source of a hang is a message that was never received, even though it is a valid one, and even though it appears to have been sent. In these situations, the problem is probably due to lost messages in the communication subsystem. This is especially true if the lost message is intermittent or varies from run to run. This is either the program's fault or the environment's fault. Before investigating the environment, you should analyze the program's *safety* with respect to MPI. A *safe* MPI program is one that does not depend on a particular implementation of MPI. You should also examine the error logs for evidence of repeated message transmissions (which usually indicate a network failure).

Although MPI specifies many details about the interface and behavior of communication calls, it also leaves many implementation details unspecified (and it does not just omit them, it specifies that they are unspecified.) This means that certain uses of MPI may work correctly in one implementation and fail in another, particularly in the area of how messages are buffered. An application may even work with one set of data and fail with another in the same implementation of MPI. This is because, when the program works, it has stayed within the limits of the implementation. When it fails, it has exceeded the limits. Because the limits are unspecified by MPI, both implementations are valid. MPI *safety* is discussed further in Chapter 5, "Creating a safe program," on page 93.

Once you have verified that the application is *MPI-safe*, your only recourse is to blame lost messages on the environment. If the communication path is IP, use the standard network analysis tools to diagnose the problem. Look particularly at **mbuf** usage. You can examine **mbuf** usage with the **netstat** command. Note that the **netstat** command is not a distributed command, which means that it applies only to the node on which you execute it.

```
$ netstat -m
```

If the **mbuf** line shows any failed allocations, you should increase the **thewall** value of your network options. You can see your current setting with the **no** command. Note that the **no** command is not a distributed command which means that it applies only to the node on which you execute it.

```
$ no -a
```

The value presented for **thewall** is in KBytes. You can use the **no** command to change this value. You will have to have root access to do this. For example,

```
$ no -o thewall=16384
```

sets **thewall** to 16 MBytes.

Message passing between lots of remote hosts can tax the underlying IP system. Make sure that you look at all the remote nodes, not just the home node. Allow lots of buffers. If the communication path is user space (US), you will need to get your system support people involved to isolate the problem.

Other reasons for the program to hang

One final cause for no output is a problem on the home node (POE is hung). Normally, a *hang* is associated with the remote hosts waiting for each other, or for a termination signal. POE running on the home node is alive and well, waiting patiently for some action on the remote hosts. If you type **<Ctrl-c>** on the POE console, you will be able to successfully interrupt and terminate the set of remote hosts. See *IBM Parallel Environment: Operation and Use, Volume 1* for information on the **poekill** command.

There are situations where POE itself can hang. Usually these situations are associated with large volumes of input or output. Remember that POE normally gets standard output from each node. If each task writes a large amount of data to standard output, it may chew up the IP buffers on the machine running POE, causing it (and all the other processes on that machine) to block and hang. The only way to know that this is the problem is by seeing that the rest of the home node has hung. If you think that POE is hung on the home node, your only solution may be to kill POE there. Press <**Ctrl-c**> several times, or use the command **kill -9**. At present, there are only partial approaches to avoiding the problem. You can allocate lots of **mbufs** on the home node, and do not make the send and receive buffers too large.

Bad output

Bad output includes unexpected error messages. After all, who expects error messages or bad results (results that are not correct)?

Error messages

You can track down the causes of error messages and correct them in parallel programs using techniques similar to those used for serial programs. One difference, however, is that you need to identify which task is producing the message, if it is not coming from all tasks. You can do this by setting the

MP_LABELIO environment variable to **yes**, or using the **-labelio yes** command line parameter. Generally, the message will give you enough information to identify the location of the problem.

You may also want to generate *more* error and warning messages by setting the **MP_EUIDEVELOP** environment variable to **yes** when you first start running a new parallel application. This will give you more information about the things that the message passing library considers errors or unsafe practices.

Bad results

You can track down bad results and correct them in a parallel program in a fashion similar to that used for serial programs. The process in the previous debugging exercise can be more complicated because the processing and control flow on one task may be affected by other tasks. In a serial program, you can follow the exact sequence of instructions that were executed and observe the values of all variables that affect the control flow. However, in a parallel program, both the control flow and the data processing on a task may be affected by messages sent from other tasks. For one thing, you may not have been watching those other tasks. For another, the messages could have been sent a long time ago. Therefore, it is very difficult to correlate a message that you receive with a particular series of events.

Debugging and threads

So far, the discussion has been about debugging normal old serial or parallel programs, but you may want to debug a threaded program (or be aware of the threads used in the library). If this is the case, there are a few things you should consider.

Before you do anything else, you first need to understand the environment in which you are working. You have the potential to create a multithreaded application, using a multithreaded library, that consists of multiple distributed tasks. As a result, finding and diagnosing bugs in this environment may require a different set of debugging techniques that you are not used to using. Here are some things to remember.

When you attach to a running program, all the tasks you selected in your program will be stopped at their current points of execution. Typically, you want to see the current point of execution of your task. This stop point is the position of the program counter, and may be in any one of the many threads that your program may create OR any one of the threads that the MPI library creates. With non-threaded programs, it was adequate to just travel up the program stack until you reached your application code (assuming you compiled your program with the **-g** option). But with threaded programs, you now need to traverse across other threads to get to your thread(s) and then up the program stack to view the current point of execution of your code.

The MPI library itself will create a set of threads to process message requests. When you attach to a program that uses the MPI library, all of the threads associated with the POE job are stopped, including the ones created and used by MPI.

For more information on the threaded MPI library, see *IBM Parallel Environment: MPI Programming Guide*.

Chapter 4. Is the program efficient?

So far, the discussions have been about getting PE working, creating message passing parallel programs, debugging problems, and debugging parallel applications. When you get a parallel program running so that it gives us the correct answer, you are done. Not necessarily. In this area, parallel programs are just like sequential programs; just because they give you the correct answer does not mean they are doing it in the most efficient manner. For a program that is relatively short running or is run infrequently, it may not matter how efficient it is. For a program that consumes a significant portion of the system resources, you need to make the best use of those resources by tuning its performance.

Tuning the performance of a parallel application

There are two approaches to tuning the performance of a parallel application.

- You can tune a sequential program and then parallelize it.

  With this approach, the process is the same as for any sequential program, and you use the same tools; **prof**, **gprof**, and **tprof**. In this case, the parallelization process must take performance into account, and should avoid anything that adversely affects it.
- You can parallelize a sequential program and then tune the result. With this approach, the individual parallel tasks are optimized together, taking both algorithm and parallel performance into account simultaneously.

Both of these techniques yield comparable results. The difference is in the tools that are used in each of the approaches, and how they are used.

Note: It may not be possible to use some tools in a parallel environment in the same way that they are used in a sequential environment. This may be because the tool requires root authority and POE restricts the root ID from running parallel jobs. Or, it may be because, when the tool is run in parallel, each task attempts to write into the same files, thus corrupting the data. **tprof** is an example of a tool that falls into both of these categories.

With either approach, you use the standard sequential tools in the traditional manner. When you tune an application and then parallelize it, observe the communication performance, how it affects the performance of each of the individual tasks, and how the tasks affect each other. For example, does one task spend a lot of time waiting for messages from another? If so, perhaps you need to rebalance the workload. Or if a task starts waiting for a message long before it arrives, perhaps it could do more algorithmic processing before waiting for the message. When an application is made parallel and then tuned, you need a way to collect the performance data in a manner that includes both communication and algorithmic information. That way, if the performance of a task needs to be improved, you can decide between tuning the algorithm or tuning the communication.

| This discussion does not deal with standard algorithmic tuning techniques. Rather,
| the discussion is about some of the ways PE can help you tune the parallel nature
| of the application, regardless of the approach you take.

How much communication is enough?

A significant factor that affects the performance of a parallel application is the balance between communication and workload. In some cases, the workload is unevenly distributed or is duplicated across multiple tasks. Ideally, you would like perfect balance among the tasks, but doing so may require additional communication that actually makes the performance worse. Sometimes it is better to have all the tasks do the same thing rather than have one do it and try to send the results to the rest.

An example of where the decision is not so clear cut is the matrix inversion program in Chapter 2, “Message passing,” on page 21. There you saw how to start making the sequential program into a parallel one by distributing the element calculation once the determinant was found. That start is actually a poor one. Part of the program is shown below.

```
/*************************************************************************
*
* Matrix Inversion Program - First parallel implementation
*
* To compile:
* mpcc -g -o inverse_parallel inverse_parallel.c
*
*************************************************************************/
     {
/* There are only 2 unused rows/columns left */

/* Find the second unused row */
for(row2=row1+1;row2<size;row2++)
  {
    for(k=0;k<depth;k++)
      {
        if(row2==used_rows[k]) break;
      }
    if(k>=depth)  /* this row is not used */
      break;
  }
assert(row2<size);

/* Find the first unused column */
for(col1=0;col1<size;col1++)
  {
    for(k=0;k<depth;k++)
      {
         if(col1==used_cols[k]) break;
      }
    if(k>=depth)  /* this column is not used */
      break;
  }
assert(col1<size);

/* Find the second unused column */
for(col2=col1+1;col2<size;col2++)
  {
    for(k=0;k<depth;k++)
      {
        if(col2==used_cols[k]) break;
      }
    if(k>=depth)  /* this column is not used */
      break;
  }
assert(col2<size);

/* Determinant = m11*m22-m12*m21 */
return matrix[row1][col1]*matrix[row2][col2]-matrix
[row1][col2]*matrix[row2][col1];
        }

        /* There are more than 2 rows/columns in the matrix being processed  */
        /* Compute the determinant as the sum of the product of each element */
        /* in the first row and the determinant of the matrix with its row   */
        /* and column removed                                                */
        total = 0;

        used_rows[depth] = row1;
        for(col1=0;col1<size;col1++)
          {
            for(k=0;k<depth;k++)
              {
                if(col1==used_cols[k]) break;
              }
             if(k<depth)  /* This column is used -- skip it*/
               continue;
            used_cols[depth] = col1;
            total += sign*matrix[row1][col1]*determinant(matrix,size,used_rows,
            used_cols,depth+1);
            sign=(sign==1)?-1:1;
          }
        return total;

  }

void print_matrix(FILE * fptr,float ** mat,int rows, int cols)
{
  int i,j;
  for(i=0;i<rows;i++)
    {
      for(j=0;j<cols;j++)
        {
          fprintf(fptr,"%10.4f ",mat[i][j]);
        }
      fprintf(fptr,"\n");
    }
}

float coefficient(float **matrix,int size, int row, int col)
{
  float coef;
  int * ur, *uc;

  ur = malloc(size*sizeof(matrix));
  uc = malloc(size*sizeof(matrix));
  ur[0]=row;
  uc[0]=col;
  coef = (((row+col)%2)?-1:1)*determinant(matrix,size,ur,uc,1);
  return coef;
}
```

The suspicion is there is a problem, and that it is not a communication bottleneck, but rather a computation problem. To illustrate this, compile the parallel matrix inversion program, **inverse_parallel.c**, with the **-pg** flag. Next, run **gprof** on the monitor files for tasks 0-7 (task 8 just collects the results so its performance is not a concern).

```
$ mpcc -g -pg -o inverse_parallel inverse_parallel.c
$ inverse_parallel -procs 9
$ gprof inverse_parallel gmon.out.[0-7] > gprof.out
```

You want to look in the output file (pick your favorite viewer, such as vi), and to get to the part we are really interested in, search for cumulative. In this case gprof produces a lot of output, so we will be skipping over a lot of it, and focusing on just a portion of what you will really see. What you are interested in is:

```
%   cumulative  self          self    total
time   seconds seconds  calls ms/call ms/call name
 38.5    2.22    2.22                        ._lapi_shm_dispatcher [1]
 26.3   3.74     1.52    72   21.11   21.11  .determinant [2]
 16.3   4.68     0.94                        ._lapi_dispatcher [6]
  5.7   5.01     0.33                        ._is_yield_queue_empty [7]
  5.0   5.30     0.29                        .LAPI__Msgpoll [8]
  2.9   5.47     0.17                        .__divu64 [9]
  0.9   5.52     0.05                        .__mcount [10]
  0.7   5.56     0.04                        ._lapi_shm_setup [11]
  0.5   5.59     0.03                        .time_base_to_time [12]
  0.3   5.61     0.02                        .__mcount [13]
  0.3   5.63     0.02                        .read_real_time [15]
  0.2   5.64     0.01   216    0.05   0.05   .std::_LFS_ON::locale::id::
                                              id(unsigned long) [16]
  0.2   5.65     0.01    32    0.31   0.31   ._alloc_pthread [17]
  0.2   5.66     0.01    24    0.42   0.42   .pthread_exit [32]
```

You see that you spend a lot of time in **determinant**, first to compute the determinant for the entire matrix and then in computing the determinant as part of computing the element values. That seems like a good place to start optimizing.

This algorithm computes the determinant of a matrix by using the determinants of the submatrices formed by eliminating the first row and a column from the matrix. The result of this recursion is that, eventually, the algorithm computes the determinants of all the 2 by 2 matrixes formed from the last two rows and each combination of columns. This is not so bad, but the same 2 by 2 matrix formed in this manner is computed n-2 times (once for each column except the 2 from which it is formed) each time a determinant is computed and there are n*(n-1)/2 such matrixes. If the 2 by 2 matrix determinants can be captured and reused, it would provide some improvements.

Not only is this a good approach for optimizing a sequential program, but parallelism capitalizes on this approach as well. Because the 2 by 2 determinants are independent, they can be computed in parallel and distributed among the tasks. Each task can take one of the columns and compute the determinants for all the matrixes formed by that column and subsequent columns. Then the determinants can be distributed among all the tasks and used to compute the inverse elements.

The following example shows only the important parts of the program.

Here is the call to partial determinant:

```
/************************************************************************
*
* Matrix Inversion Program - First optimized parallel version
*
* To compile:
* mpcc -g -o inverse_parallel_fast inverse_parallel_fast.c
*
*************************************************************************/

  /* Compute determinant of last two rows */
  pd = partial_determinant(matrix,rows);
  /* Everyone computes the determinant (to avoid message transmission) */
  determ=determinant(matrix,rows,used_rows,used_cols,0,pd);
```

And here is the partial determinant call:

```
/* Compute the determinants of all 2x2 matrixes created by combinations */
/* of columns of the bottom 2 rows                                       */
/* partial_determinant[i] points to the first determinant of all the 2x2*/
/* matrixes formed by combinations with column i.  There are n-i-1     */
/* such matrixes (duplicates are eliminated)                           */
float **partial_determinant(float **matrix,int size)
{
  int col1, col2, row1=(size-2), row2=(size-1);
  int i,j,k;
  int terms=0;
  float **partial_det,  /* pointers into the 2x2 determinants*/
                        /* by column                         */
        *buffer,        /* the 2x2 determinants              */
        *my_row;        /* the determinants computed by this */
                        /* task                              */
  int * recv_counts, * recv_displacements; /* the size and offsets for the */
                                        /* determinants to be received from*/
                                           /* the other tasks        */

  terms = (size-1)*(size)/2;  /* number of combinations of columns */

  /* Allocate work areas for partial determinants and message passing, */
  partial_det = (float **) malloc((size-1)*sizeof(*partial_det));
  buffer      = (float *)  malloc(terms*sizeof(buffer));
  my_row      = (float *)  malloc((size-me-1)*sizeof(my_row));
  recv_counts = (int *)    malloc(tasks*sizeof(*recv_counts));
  recv_displacements = (int *) malloc(tasks*sizeof(*recv_displacements));

  /* the tasks after the column size - 2 don't have to do anything */
  for(i=tasks-1;i>size-2;i--)
    {
        recv_counts[i]=0;
        recv_displacements[i]=terms;
    }
  /* all the other tasks compute the determinants for combinations */
  /* with its column                                               */
  terms--;
  for(i=size-2;i>=0;i--)
    {
        partial_det[i]=&(buffer[terms]);
        recv_displacements[i]=terms;
        recv_counts[i]=size-i-1;
        terms-=(size-i);
    }
  for(j=0;j<(size-me-1);j++)
    {
      my_row[j]=matrix[row1][me]*matrix[row2][me+j+1]
      -matrix[row1][me+j+1]*matrix[row2][me];
    }

  /* Now everybody sends their columns determinants to everybody else */
  /* Even the tasks that did not compute determinants will get the    */
  /* results from everyone else (doesn't sound fair, does it?)        */
  MPI_Allgatherv(my_row,
               ((size-me-1)>0)?(size-me-1):0,
               MPI_REAL,
               buffernts,
               recv_displacements,
               MPI_REAL,MPI_COMM_WORLD);

  /* Free up the work area and return the array of pointers into the */
  /* determinants                                                    */
  free(my_row);
  return partial_det;
}
```

The question is whether the cost of the additional communication offsets the advantage of computing the 2 by 2 determinants in parallel. In this example, it may not be because the small message sizes (the largest is three times the size of a float). As the matrix size increases, the cost of computing the 2 by 2 determinants will increase with the square of n (the size of the matrix) but the cost of computing the determinants in parallel will increase with n (each additional dimension increases the work of each parallel task by only one additional 2 by 2 matrix) so, eventually, the parallel benefit will offset the communication cost.

Tuning the performance of threaded programs

There are some things you need to consider when you want to get the maximum performance out of the program.

Note: The PE implementation of MPI (PE MPI) is threadsafe.

- Two environment variables affect the overhead of an MPI call in the threaded library:
  - **MP_SINGLE_THREAD=[no|yes]**
  - **MP_EUIDEVELOP=[no|yes|deb|min]**

| For a program that has only one MPI communication thread, you can set the environment variable **MP_SINGLE_THREAD** to **yes** before running. This will avoid some locking which is otherwise required to maintain consistent internal MPI state. The program may have other threads that do computation or other work, as long as they do not make MPI calls. Note that the implementation of MPI I/O and MPI one-sided communication is thread-based, and that these facilities may not be used when **MP_SINGLE_THREAD** is set to **yes**. Set **MP_SINGLE_THREAD** to **yes** only if you are certain the application has only one thread making MPI calls. If there are two or more threads calling MPI, you will not get any warning from MPI and will experience race conditions that lead to unpredictable errors. Applications that pass large numbers of tiny messages may see measurable performance gains from setting **MP_SINGLE_THREAD** to **yes**. Most applications will see no measurable improvement.

The **MP_EUIDEVELOP** environment variable lets you control how much checking is done when you run the program. Eliminating checking altogether (setting **MP_EUIDEVELOP** to **min**) provides performance (latency) benefits, but may cause critical information to be unavailable if the executable hangs due to message passing errors. For more information on **MP_EUIDEVELOP** and other POE environment variables, see *IBM Parallel Environment: Operation and Use, Volume 1*.

- Programs (threaded or non-threaded) that use the threaded MPI library can be profiled by using the **-pg** flag on the compilation and linking step of the program.

  The profile results (gmon.out) will contain only a summary of the information from all the threads per task together. Viewing the data using gprof or Xprofiler is limited to showing only this summarized data on a per task basis, not per thread.

  Note: AIX supports thread profiling. There are changes to the format, content, and naming of the profiling output files produced by prof and gprof. For additional details, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*.

For more information on profiling, see *AIX 5L Version 5.3: Performance Tools Guide and Reference*.

Why is this so slow?

You have a serial program and you want it to execute faster. In this situation, it is best not to jump into parallelizing the program right away. Instead, you start by tuning the serial algorithm.

The program in this next example approximates the two-dimensional Laplace equation and uses a 4-point stencil.

The algorithm is very straightforward. For each array element, you will assign that element the average of the four elements that are adjacent to it (except the rows and columns that represent the boundary conditions of the problem).

You may find it helpful to refer to *In Search of Clusters* by Gregory F. Pfister for more information on this problem and how to parallelize it.

The 4-point stencil program is central to this entire discussion, so you may want to spend some time to understand how it works.

The first step is to compile the serial program. However, before you do this, be sure you have a copy of **stencil.dat** in the program directory, or run the **init** program to generate one. Once you have done this, you can compile the serial program with the **xlf** command:

```
$ xlf -O2 naive.f -o naive
```

Next, you need to run the program and collect some information to see how it performs. You can use the UNIX **time** command to do this:

```
$ time naive
```

Table 2 shows the result:

Table 2. Results of program 'naive'

Program Name	Tasks	Wallclock Time	Array Size per Task
naive	1 (single processor)	11 min. 1.94 sec.	1000 by 1000

The execution time appearing in Table 2 was obtained with an earlier version of PE MPI on an IBM eServer clustered 1600 server. The execution times may be different, depending on the system that you are using, but the concepts for improving an application to reduce execution time are unchanged.

Looking at these results, there is room for improvement, especially if you scale the problem to a much larger array. So, how can you improve the performance?

Profile it

The first step in tuning the program is to find the areas within the program that execute most of the work. Locating these compute-intensive areas within the program lets you focus on the areas that give you the most benefit from tuning. The best way to find them is to *profile* the program.

Profile the program using Xprofiler

When you profile your program, you need to compile it with the **-pg** flag to generate profiling data. Note that the `-O2` flag is a capital letter O followed by the number 2:

```
$ xlf -pg -O2 naive.f -o naive
```

The **-pg** flag compiles and links the executable so that when you run the program, the performance data gets written to output.

Now that you have compiled your program with the **-pg** flag, run it again to see what you get:

```
$ naive
```

This generates a file called **gmon.out** in the current working directory. you can look at the contents of **gmon.out** with the Xprofiler profiling tool. This tool is part of the AIX operating system. For more information about Xprofiler, see *AIX 5L Version 5.3: Performance Tools Guide and Reference*.

AIX supports thread profiling and, in doing so, has changed the format and name of the profiling output files. For more information on the default profiling output file names, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2*.

To start Xprofiler, you will use the **xprofiler** command, like, this:

```
$ xprofiler naive gmon.out
```

The Xprofiler main window appears, and in this window you will see the **function call tree**. The function call tree is a graphical representation of the functions within the application and their interrelationships. Each function is represented by a green, solid-filled box called a *function box*. In simple terms, the larger this box, the greater percentage of the total running time it consumes. So, the largest box represents the function doing the most work. The calls between functions are represented by blue arrows drawn between them *call arcs*. The arrowhead of the call arc points to the function that is being called. The function boxes and call arcs that belong to each library in the application appear within a fenced-in area called a *cluster box*. For the purposes of this discussion, you will remove the cluster boxes from the display.

PLACE
the mouse cursor over the Filter menu.

CLICK
the left mouse button

The Filter menu appears.

SELECT
the **Hide All Library Calls** option.

The library calls disappear from the function call tree.

PLACE
the mouse cursor over the Filter menu.

CLICK
the left mouse button.

The Filter menu appears.

SELECT
the **Uncluster Functions** option.

The functions expand to fill the screen.

Locate the largest function box in the function call tree. You can get the name of the function by looking a little more closely at it:

PLACE
the mouse cursor over the View menu.

The View menu appears.

PLACE
the mouse cursor over the **Overview** option.

CLICK
the left mouse button.

The Overview Window appears.

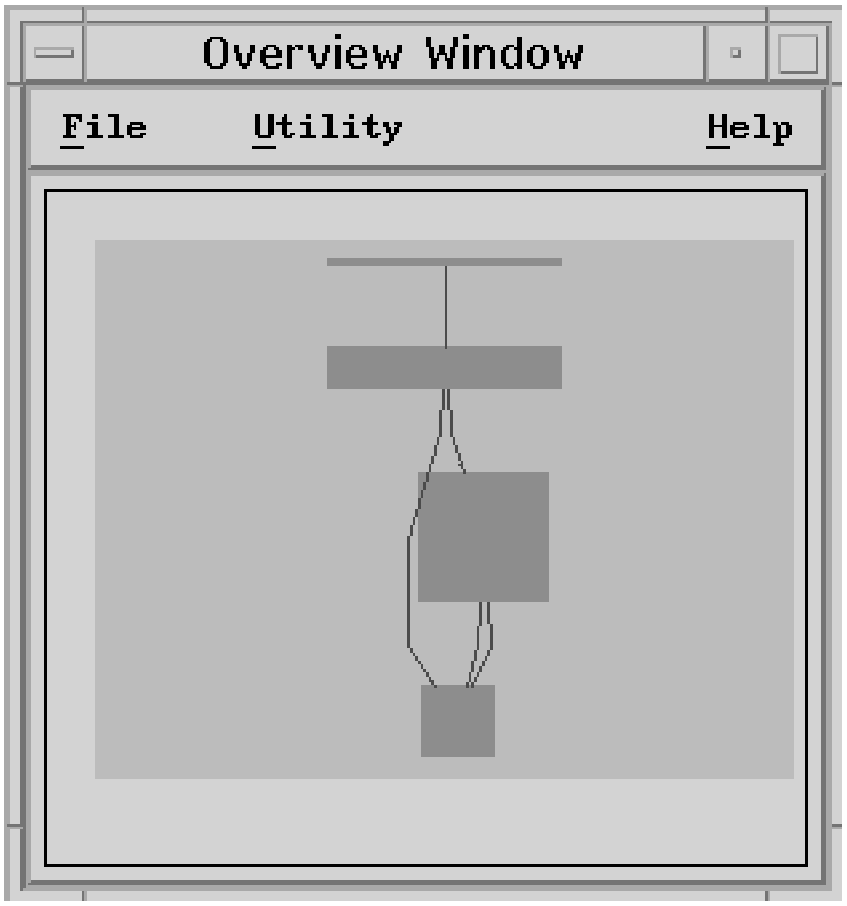


Figure 1. Overview window

The Overview Window includes a light blue highlight area that lets you zoom in and out of specific areas of the function call tree. To take a closer look at the largest function of naive:

PLACE
the mouse cursor over the lower left corner of the blue highlight area. You know that the cursor is over the corner when the cursor icon changes to a right angle with an arrow pointing into it.

PRESS and HOLD
the left mouse button, and drag it diagonally upward and to the right (toward the center of the sizing box) to shrink the box. When it is about half its original size, release the mouse button.

The corresponding area of the function call tree, in the main window, appears magnified.

If the largest function was not within the highlight area, it did not get magnified. If this was the case, you will need to move the highlight area:

PLACE
the cursor over the highlighted area.

PRESS and HOLD
the left mouse button.

DRAG the highlight area, using the mouse, and place it over the largest function. Release the mouse button.

The largest function appears magnified in the function call tree.

Just below the function is its name, so you can now see that most of the work is being done in the **compute_stencil()** subroutine. This subroutine is where you should focus your attention.

It is important to note that the programming style you choose can influence the program's performance just as much as the algorithm you use. In some cases, this will be clear by looking at the data you collect when the program executes. In other cases, you will know this from experience. There are many books that cover the subject of code optimization, many of which are extremely complex.

The goal here is not to use every optimization trick but to focus on some basic techniques that can produce the biggest performance boost for the time and effort spent.

Profile the program using the Performance Collection Tool

The best way to begin is to look at your use of memory (including hardware data cache) as well as what you are doing in the critical section of your code. To do this, use the Performance Collection Tool to count the number of cache misses. The fewer the number of cache misses, the better the performance of your code will be.

When you profile your program using PCT, you need to compile it with the required **-g** flag to generate profiling data. You can also include the optional **-o** flag to specify an output file:

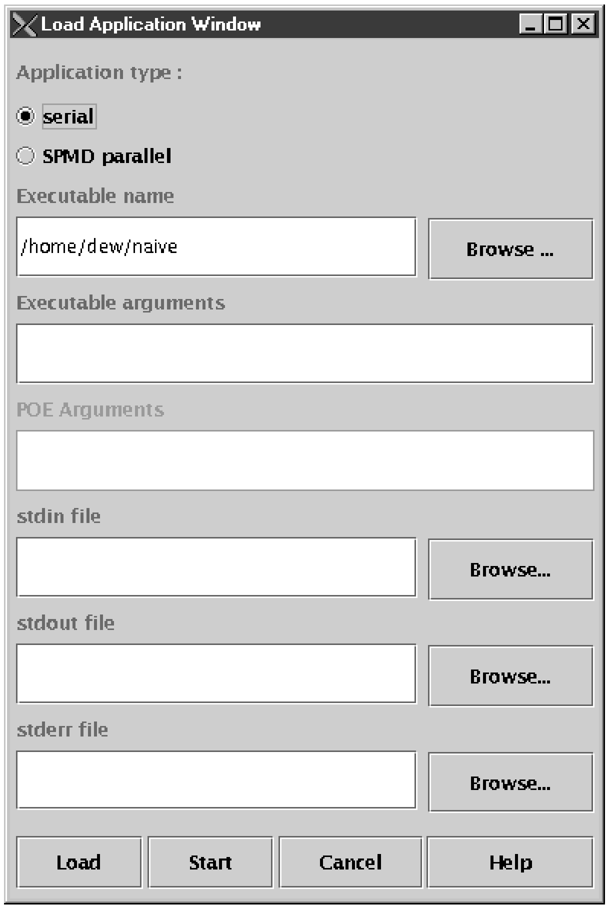
```
$ xlf -g -o naive naive.f
```

Once you have generated the profiling data, you can use PCT to examine the data in detail.

TYPE **pct** to start up the Performance Collection Tool graphical user interface. From the main window, you are prompted to either load and start an application or connect to one that is already running.

SELECT
the **Load a new application** option and click on **OK**.

The **Load Application** window opens and you are prompted to select the application you want to load.



Application type : [✓ serial] SPMD parallel

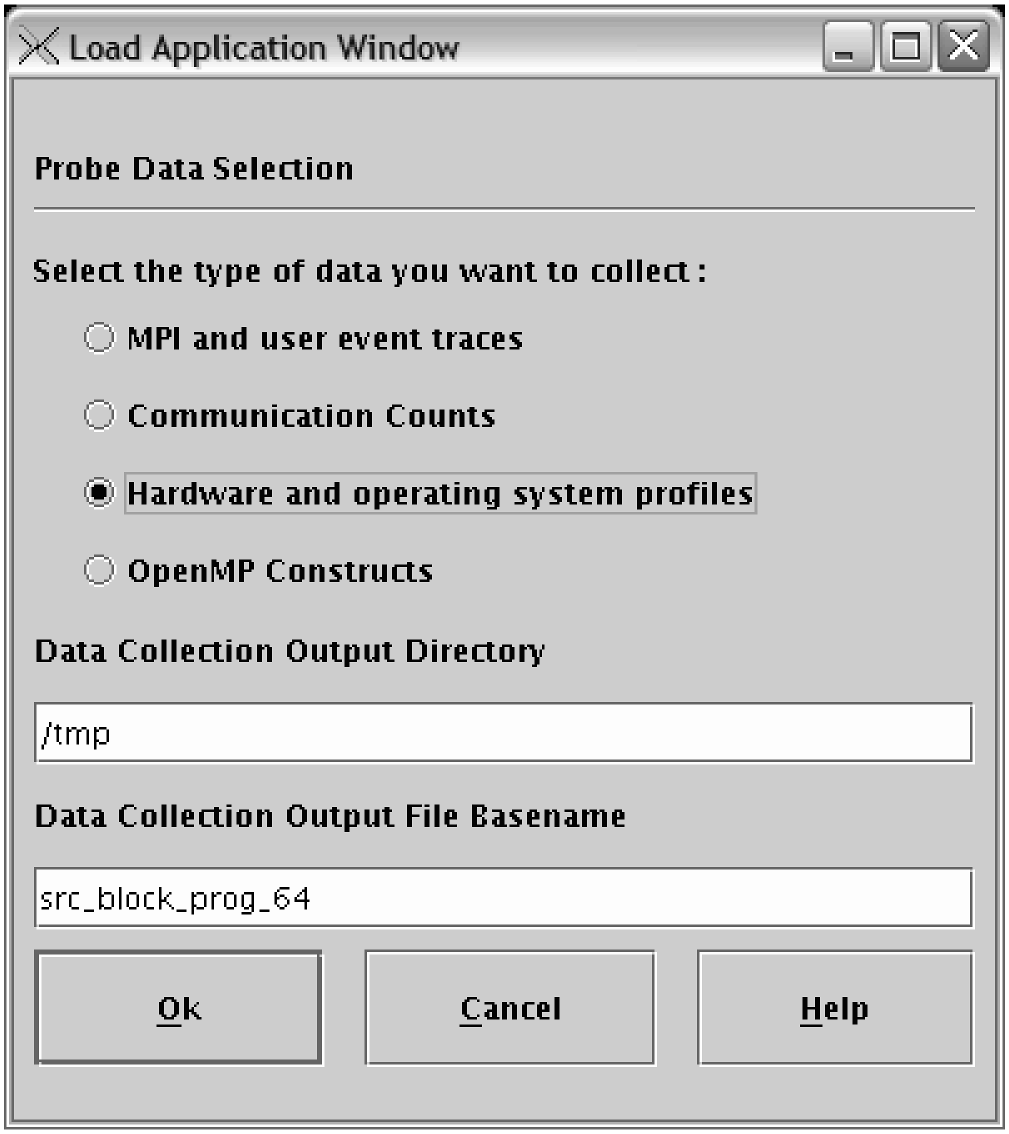
Executable name: /home/dew/naive

Figure 2. Load application window

CLICK
: the **Browse** button next to the **Executable Name** field and select the **naive** program and identify it as a serial application.

CLICK
: the **Load** button to load the application.

The **Probe Data Selection** window opens.



Probe Data Selection

Select the type of data you want to collect :

- ○ MPI and user event traces
- ○ Communication Counts
- ● Hardware and operating system profiles
- ○ OpenMP Constructs

Data Collection Output Directory: /tmp

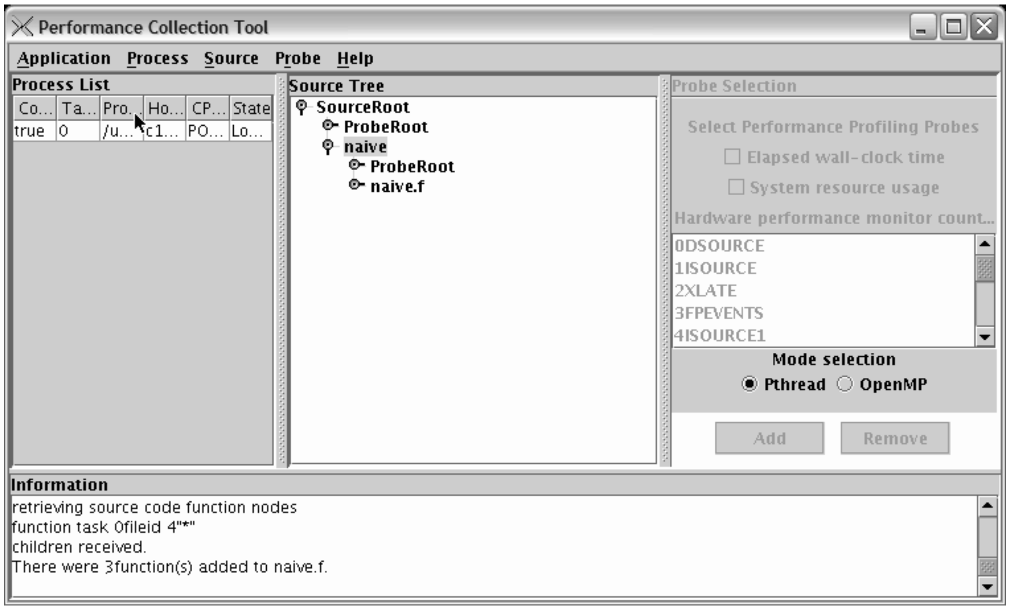
Data Collection Output File Basename: src_block_prog_64

Figure 3. Probe data selection window

SELECT
: the type of data you want to collect. Select the **Hardware and operating system profiles** option.

SPECIFY
: the directory and base name for the output file and click **OK**. Note that the base name you specify will have a **.cdf** suffix and a task number suffix appended to it.

  The main window comes to the foreground and the source tree for the **naive** executable is expanded.



```
retrieving source code function nodes
function task 0fileid 4"*"
children received.
There were 3function(s) added to naive.f.
```

Figure 4. Source tree window

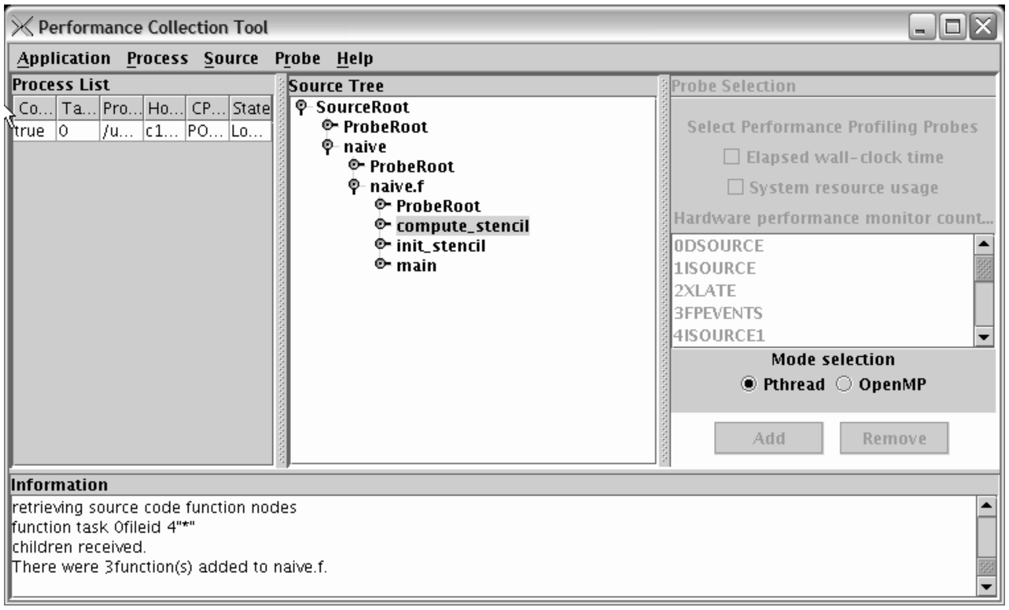
SELECT
the **naive** task from the Process List.

SELECT
the **naive_f** function to expand it.

SELECT
the **compute_stencil()** subroutine from the **naive.f** file in the source tree.

SELECT
the hardware counter probe to collect cache information. You will want to select the `L1` option to display level one information. For example, the option you select may look like:

```
2 L1_TLB
```



```
retrieving source code function nodes
function task 0fileid 4"*"
children received.
There were 3function(s) added to naive.f.
```

Figure 5. Process list, source tree, and probe selection window

CLICK
: the **Add** button. If you look at the **compute_stencil()** subroutine in the source tree, you will see that a Probe ID has been added.

SELECT
: **Application** → **Start** from the menu bar to run the program.

  When the application program has finished executing, the **Target Application Exited** window appears. Click on the **OK** button to exit PCT.

Profile the program using the Profile Visualization Tool

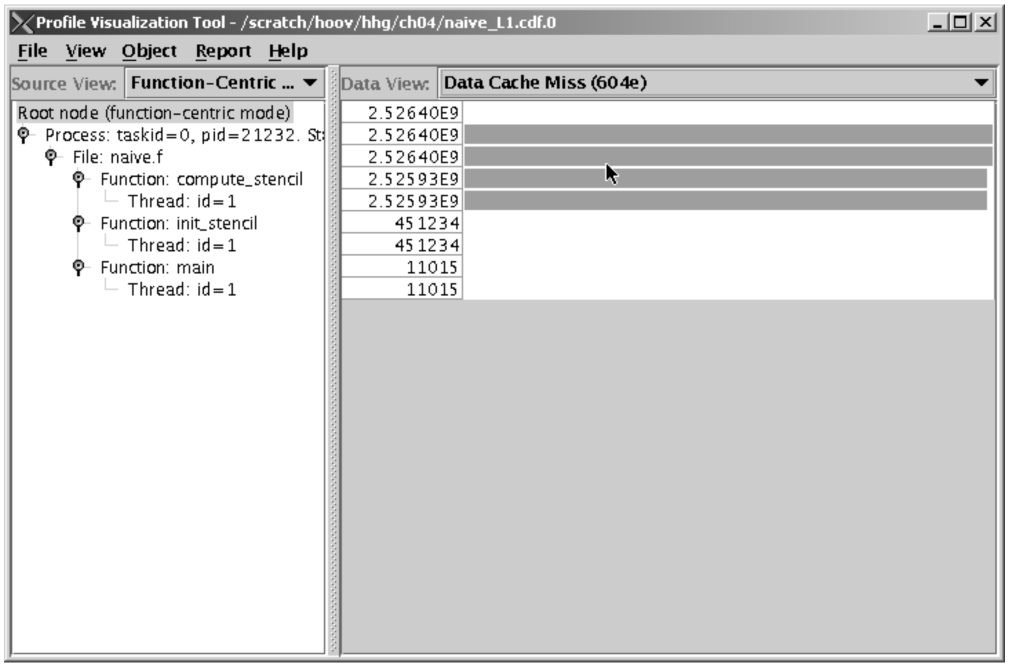
Now that you have collected your data on cache misses, you want to be able to view it and you can do that using the Profile Visualization Tool (PVT). PCT generates a NetCDF file (Network Common Data File) which you can view using PVT.

TYPE **pvt** to start up the Profile Visualization Tool.

SELECT
: **File** → **Load** from the menu bar to select and load the CDF file. Locate the CDF file that was generated from PCT from the list of files that appears and select it.

CLICK
: the **Open** button to load the file.

SELECT
: **View** → **Expand All** to expand the tree to view the function **compute_stencil()**


Profile Visualization Tool - /scratch/hoov/hhg/ch04/naive_L1.cdf.0
File View Object Report Help
Source View: Function-Centric ...
Data View: Data Cache Miss (604e)

```
Root node (function-centric mode)      2.52640E9
  Process: taskid=0, pid=21232. St     2.52640E9
    File: naive.f                      2.52640E9
      Function: compute_stencil        2.52593E9
        Thread: id=1                   2.52593E9
      Function: init_stencil           451234
        Thread: id=1                   451234
      Function: main                   11015
        Thread: id=1                   11015
```

Figure 6. Data view area

CLICK
the **Function Call Count** option in the pulldown menu located in the top right side of the Data View area. Select the **Data cache miss** option to view the number of cache misses for the function **compute_stencil**. The amount of L1 cache misses for each function are listed in the Data View window area.

Let's look at your code:

```
iter_count = 0
100  CONTINUE
local_err = 0.0
iter_count = iter_count + 1

DO i=1, m-2
DO j=1, n-2
old_value = stencil(i,j)

stencil(i,j) = ( stencil(i-1, j ) +
1                          stencil(i+1, j ) +
2                          stencil( i ,j-1) +
3                          stencil( i ,j+1) ) / 4
local_err = MAX(local_err,ABS(old_value-stencil(i,j)))
END DO
END DO
IF(MOD(iter_count,100).EQ.0)PRINT *, iter_count, local_err
IF (close_enough.LT.local_err) GOTO 100
PRINT *, "convergence reached after ", iter_count, " iterations."
```

By looking at the two DO loops above, you can see that your compute subroutine is traversing your array first across rows, and then down columns. This program must have been written by some alien being from the planet *C* because Fortran arrays are stored in *column* major form rather than *row* major form.

The first improvement you should make is to reorder your loops so that they traverse down columns rather than across rows. This should provide a reasonable performance boost. Note that it is not always possible to change the order of loops; it depends on the data referenced within the loop body. As long as the values used in every loop iteration do not change when the loops are reordered, then it is safe to change their order. In the example it was safe to reorder the loops, so here is what the revised program looks like. Notice that only the order of the loops was swapped.

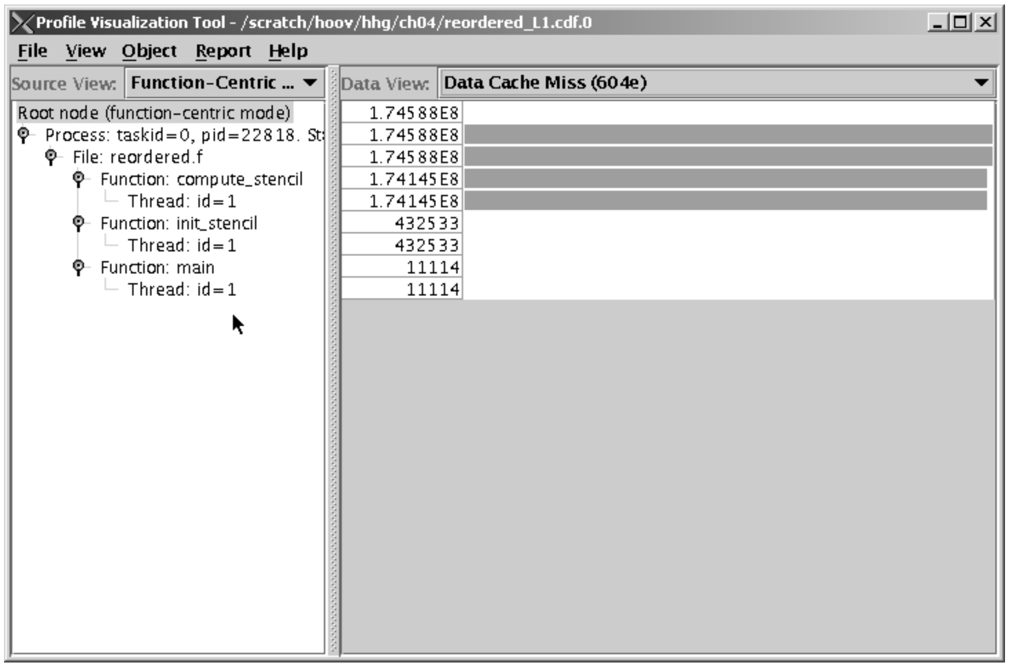
```
DO j=1, n-2
DO i=1, m-2
old_value = stencil(i,j)
```

The second thing you should look at is the type of work that is being done in your loop. If you look carefully, you will notice that the MAX and ABS subroutines are called in each iteration of the loop, so you should make sure these subroutines are compiled inline. Because these subroutines are intrinsic to your Fortran compiler, this is already done for us.

```
$ xlf -O2 reordered.f -o reordered
```

In the last scenario, you ran the naive program. You should now run the same scenario using the reordered program to more accurately compare the cache misses. You should see that the number of cache misses for reordered has decreased, thereby increasing the program's efficiency.

If you run the previous scenario again using the reordered subroutine, you notice that the cache misses are lower:


Profile Visualization Tool - /scratch/hoov/hhg/ch04/reordered_L1.cdf.0
File View Object Report Help
Source View: Function-Centric ...
Data View: Data Cache Miss (604e)

```
Root node (function-centric mode)            1.74588E8
Process: taskid=0, pid=22818. St             1.74588E8
  File: reordered.f                          1.74588E8
    Function: compute_stencil                1.74145E8
      Thread: id=1                           1.74145E8
    Function: init_stencil                      432533
      Thread: id=1                              432533
    Function: main                               11114
      Thread: id=1                               11114
```

Figure 7. Data view area (fewer cache misses showing)

As before, you need to time your run, like this:

```
$ time reordered
```

And here are the results as compared to the original naive version:

Table 3. Comparison of programs 'naive' and 'reordered'

Program Name	Tasks	Wallclock Time	Array Size per Task
naive	1 (single processor)	11 min. 1.94 sec.	1000 by 1000
reordered	1 (single processor)	5 min. 35.38 sec.	1000 by 1000

The execution times appearing in Table 3 were obtained with an earlier version of PE MPI on an IBM eServer clustered 1600 server. The execution times may be different, depending on the system that you are using, but the concepts for improving an application to reduce execution time are unchanged.

As you can see by the results, with just a small amount of analysis, you doubled performance. And you have not even considered parallelism yet. However, this still is not the performance that you want, especially for very large arrays (the CPU time is good, but the elapsed time is not).

Parallelize it

Now feeling confident that your serial program is reasonably efficient, you should look at ways to parallelize it. There are many ways to parallelize a program, but the two most commonly used techniques are functional decomposition and data decomposition. You will focus on data decomposition.

How do you *decompose* your data? Start by dividing the work across the processors. Each task will compute a section of an array, and each program will solve 1/*n* of the problem when using *n* processors.

Here is the algorithm:

- First, divide up the array space across each processor (each task will solve a subset of the problem independently).
- Second, loop:
  - exchange shared array boundaries
  - solve the problem on each sub array
  - share a global max

  until the global max is within the tolerance.

The section of code for your algorithm looks like this:

```
      iter_count = 0
 100  CONTINUE
         local_err = 0.0
         iter_count = iter_count + 1
         CALL exchange(stencil, m, n)

         DO j=1, n-2
            DO i=1, m-2
               old_value = stencil(i,j)

               stencil(i,j) = ( stencil(i-1, j ) +
     1                          stencil(i+1, j ) +
     2                          stencil( i ,j-1) +
     3                          stencil( i ,j+1) ) / 4

               local_err = MAX(local_err,ABS(old_value-stencil(i,j)))
            END DO
         END DO
        CALL MPI_Allreduce(local_err, global_error, 1, MPI_Real,
     1      MPI_Max, MPI_Comm_world, ierror)

        IF(MOD(iter_count,100).EQ.0)PRINT *, iter_count, global_error
      IF (close_enough.LT.global_error) GOTO 100
      PRINT *, "convergence reached after", iter_count, "iterations."
```

Now, let's compile your parallelized version:

```
$ mpxlf -02 chaotic.f -o chaotic
```

Next, let's run it and look at the results:

```
$ export MP_PROCS=4
$ export MP_LABELIO=yes
$ time poe chaotic
```

Table 4. Comparison of programs 'naive', 'reordered', and 'chaotic'

Program Name	Tasks	Wallclock Time	Array Size per Task
naive	1 (single processor)	11 min. 1.94 sec.	1000 by 1000
reordered	1 (single processor)	5 min. 35.38 sec.	1000 by 1000
chaotic	4 (processors)	2 min. 4.58 sec.	500 by 500

| The execution times appearing in Table 4 were obtained with an earlier version of
| PE MPI on an IBM eServer clustered 1600 server. The execution times may be
| different, depending on the system that you are using, but the concepts for
| improving an application to reduce execution time are unchanged.

The previous results show that you more than doubled performance by parallelizing your program. Since you divided up the work between four processors, you expected your program to execute four times faster. Why did it not do so? This could be due to one of several factors that tend to influence overall performance:
- Message passing overhead
- Load imbalance
- Convergence rates

Right now you need to ask something more important; does the parallel program get the same answer?

The algorithm you chose gives us a *correct* answer, but as you will see, it does not give us the *same* answer as your serial version. In practical applications, this may be acceptable. In fact, it is very common for this to be acceptable in Gauss/Seidel chaotic relaxation. But what if it is not acceptable? How can you tell? What methods or tools can be used to help us diagnose the problem and find a solution?

Wrong answer!

You have now invested all this time and energy in parallelizing your program using message passing, so why can you not get the same answer as the serial version of the program? This is a problem that many people encounter when parallelizing applications from serial code and can be the result of algorithmic differences, program defects, or environment changes.

Both the serial and parallel versions of your program give correct answers based on the problem description, but that does not mean they both cannot compute different answers! Let's examine the problem more closely by running the **chaotic.f** program under the **pdbx** debugger:

```
$ pdbx chaotic
```

By looking at the main program, you can see that both versions of your program (**reorder.f** and **chaotic.f**) read in the same data file as input. And after you initialize your parallel environment, you can see that the **compute_stencil** subroutine performs exactly the same step to average stencil cells.

Run each version under the control of the debugger to view and compare the results of your arrays.

With this test, you will be looking at the upper left quadrant of the entire array. This allows us to compare the array subset on task 0 of the parallel version with the same subset on the serial version.

Here is the serial (reordered) array and parallel (chaotic) array stencils:

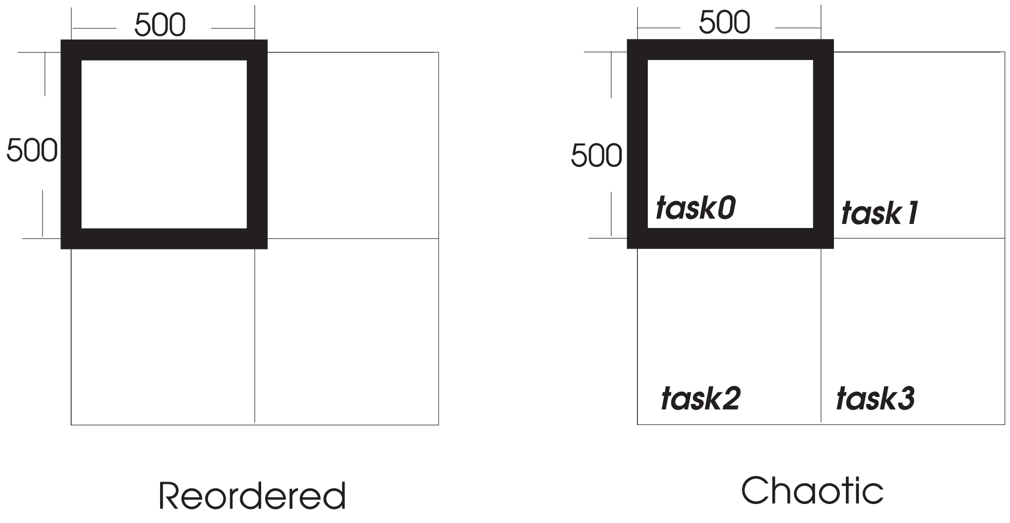


Figure 8. Serial and parallel array stencils

In **chaotic.f**, set a breakpoint within the call **compute_stencil** at line 168.

```
pdbx(all) stop at 168
all:[0] stop at "chaotic.f":168
```

After you do this, all tasks should have a breakpoint set at line 168.

Continue to execute the program up to the breakpoints. The program counter should now be positioned at line 168.

```
pdbx(all) cont
   0: initializing the array.
   0: computing the stencil.
   0: 100 1.397277832
   1: 100 1.397277832
   ...
   ...
   1:[6] stopped in compute_stencil at line 168 in file "chaotic.f" ($t1)
   1:  168         PRINT *, "convergence reached after", iter_count, "iterations."
   2:[6] stopped in compute_stencil at line 168 in file "chaotic.f" ($t1)
   2:  168         PRINT *, "convergence reached after", iter_count, "iterations."
   3:[6] stopped in compute_stencil at line 168 in file "chaotic.f" ($t1)
   3:  168         PRINT *, "convergence reached after", iter_count, "iterations."
   0:[6] stopped in compute_stencil at line 168 in file "chaotic.f" ($t1)
   0:  168         PRINT *, "convergence reached after", iter_count, "iterations."
```

Next, you will need to examine the array *stencil*. Switch the context to task 0, then print the 499th row of the array:

```
pdbx print stencil(499,1..10)

0:(499,1) = 8.00365734
0:(499,2) = 15.9983482
0:(499,3) = 23.9780369
0:(499,4) = 31.9367294
0:(499,5) = 39.8684845
0:(499,6) = 47.7674294
0:(499,7) = 55.6277695
0:(499,8) = 63.4438095
0:(499,9) = 71.2099609
0:(499,10) = 78.9207458      ...
```

Let's take a close look at the data of each.

Here is the **reordered** data:

```
(row, col)
 (499,1)    (499,2)      (499,3)      (499,4)      (499,5)      (499,6)
 8.00365734  15.9983482   23.9780369   31.9367294   39.8684845   47.7674294
 (499,7)    (499,8)      (499,9)      (499,10)
  55.6277695 63.4438095   71.2099609   78.9207458
```

Here is the **chaotic** data:

```
(row, col)
(499,1)     (499,2)      (499,3)      (499,4)      (499,5)      (499,6)
 8.04555225  16.0820065   24.1032257   32.1031151   40.0756378   48.0148277
(499,7)     (499,8)      (499,9)      (499,10)
 55.9147987  63.7697601   71.5740356   79.3220673
```

After looking at the data, you see that your answers are definitely similar, but different. Why? You can blame it on a couple of things, but it is mostly due to the chaotic nature of your algorithm. By looking at how the average is computed in the serial version of your program, you can see that within each iteration of your loop, two array cells are from the old iteration and two are from new ones.

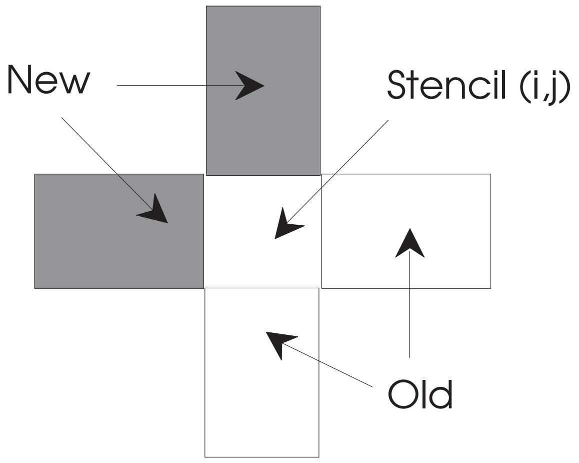


Figure 9. How the average is computed in a 4-point stencil

Another factor is that the north and west borders contain old values at the beginning of each new sweep for all tasks except the northwest corner. The serial version would use **new** values in each of those quadrants instead of old values. In the parallel version of your program, this is true for the interior array cells but not for your shared boundaries. For more information, you may find *In Search of Clusters* by Gregory F. Pfister, Prentice Hall, 1998, helpful.

OK, now that you know why you get different answers, is there a fix?

Here’s the fix!

So, you have a serial and parallel program that do not give you the same answers. One way to fix this is to skew the processing of the global array. You skew the processing of the array, computing the upper left process coordinate first, then each successive diagonal to the lower right process coordinate. Each process sends the east and south boundary to its neighboring task.

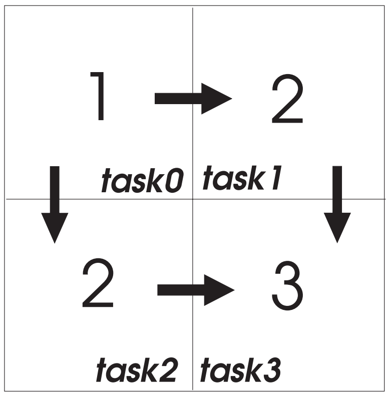


Figure 10. Sequence of array calculation

The only thing you need to modify in your new program is the message passing sequence. Prior to the **compute_stencil()** subroutine, each task receives boundary cells from its north and west neighbors. Each task then sends its east and south boundary cells to its neighbor. This guarantees that the array cells are averaged in the same order as in your serial version.

Here is your modified (skewed) parallel program. It is called **skewed.f**.

```
      iter_count = 0
100   CONTINUE
         local_err = 0.0
         iter_count = iter_count + 1
         CALL exch_in(stencil, m, n)

         DO j=1, n-2
            DO i=1, m-2
               old_value = stencil(i,j)

               stencil(i,j) = ( stencil(i-1, j ) +
     1                          stencil(i+1, j ) +
     2                          stencil( i ,j-1) +
     3                          stencil( i ,j+1) ) / 4

               local_err = MAX(local_err,ABS(old_value-stencil(i,j)))
            END DO
         END DO

         CALL exch_out(stencil, m, n)
         CALL MPI_Allreduce(local_err, global_error, 1, MPI_Real,
     1      MPI_Max, MPI_Comm_world, ierror)
```

```
      IF(MOD(iter_count,100).EQ.0)PRINT *, iter_count, global_error
   IF (close_enough.LT.global_error) GOTO 100
   PRINT *, "convergence reached after", iter_count, "iterations."
```

Now let's run this new version and look at the results:

```
$ time poe skewed
```

Table 5. Comparison of programs 'naive', 'reordered', 'chaotic', and 'skewed'

Program Name	Tasks	Wallclock Time	Array Size per Task
naive	1 (single processor)	11 min. 1.94 sec.	1000 by 1000
reordered	1 (single processor)	5 min. 35.38 sec.	1000 by 1000
chaotic	4 (processors)	2 min. 4.58 sec.	500 by 500
skewed	4 (processors)	4 min. 41.87 sec.	500 by 500

The execution times appearing in Table 5 were obtained with an earlier version of PE MPI on an IBM eServer clustered 1600 server. The execution times may be different, depending on the system that you are using, but the concepts for improving an application to reduce execution time are unchanged.

If you do the same array comparison again, you can see that you do indeed get the same results. But, of course, nothing is that easy. By correcting the differences in answers, you slowed down execution significantly, so the hidden cost here is *time*. Now what do you do?

It's still not fast enough!

You have obtained the right answers now, but you still want your program to move faster. Look at your new code to see what other techniques you can use to speed up execution. You will look at:
- Convergence rates (total number of iterations)
- Load balance
- Synchronization/communication time.

One way to further analyze your program is to use the Argonne National Laboratory's Jumpshot tool. Using the PE Benchmarker **traceTOslog2** utility, you can generate a SLOG2 file which you can then load into Jumpshot and use to determine how you can get your program to run faster. You are going to use Jumpshot to determine the effectiveness of the program's message passing characteristics.

The traceTOslog2 command, which is used to invoke the PE traceTOslog2 utility, is provided as part of the slog2 package available from Argonne National Laboratory.

Step 1 - Determine which SLOG file to generate

PE has the ability to produce two types of SLOG files (called SLOG and SLOG2), which have incompatible formats. IBM recommends you produce SLOG2 files, and the following examples illustrate this. The SLOG and SLOG2 files must be used with the correct utilities, according to these rules:
- SLOG files are created by the PE **slogmerge** utility, and are passed as input to the Jumpshot-3 utility.
- SLOG2 files are created by the PE **traceTOslog2** utility and passed to the Jumpshot-4 utility.

Both Jumpshot-3 and Jumpshot-4 are public domain programs developed by Argonne National Laboratory. If you are not sure which one is installed on the system, ask the administrator.

For information about SLOG2 files, see **http://www-unix.mcs.anl.gov/perfvis/software/log_format/index.htm#SLOG-2**. For information about Jumpshot-4, see **http://www-unix.mcs.anl.gov/perfvis/software/viewers/index.htm#Jumpshot-4**. For information about performance visualization from Argonne National Laboratory, see **http://www.mcs.anl.gov/perfvis**.

Step 2 - Link program with the library that created MPI trace files

Before analyzing the program using Jumpshot, you must link the program with the library that creates the MPI trace files used in the analysis. You do this by setting the **MP_UTE** environment variable to YES before compiling the program. Assuming you are using ksh, issue the command **export MP_UTE=YES** before compiling the program. Once you set the environment variable, it remains set for the duration of the login session.

Step 3 - Gather performance data to AIX trace file

TYPE **pct** to start up the Performance Collection Tool graphical user interface. From the Welcome window, you are prompted to either load and start an application or to connect to one that is already running.

SELECT
the **Load a new application** option and click on **OK**.

The **Load Application** window opens and you are prompted to select the application you want to load.

CLICK
the **Browse** button next to the **Executable Name** field and select the **chaotic** program and identify it as an SPMD application.

TYPE the POE arguments in the **POE Arguments** field. For example, the following argument specifies that you are running a 4–way parallel job:

```
-procs 4
```

CLICK
the **Load** button to load the application.

The **Probe Data Selection** window opens.

SELECT
the type of data you want to collect. You want to select the MPI and user event traces option.

SPECIFY
the directory and base name for the output file. In this scenario, you are using the base name *mytrace*. Then click on **OK**.

The main window appears again with the source tree for the **skewed** executable expanded.

SELECT
Process → **Select All Tasks**.

SELECT
the **chaotic.f ()** subroutine from the source tree.

SELECT
the **All MPI events** to collect trace information from the Probe Selection area on the side of the main window.

CLICK
the **Add** button.

SELECT
Application → **Start** from the menu bar to run the program.

When the application program has finished executing, the **Target Application Exited** window appears. Click on the **OK** button to exit PCT.

Step 4 - Convert AIX trace file to UTE interval files

You have successfully collected data on message passing that now exists in a standard AIX trace file. To view and analyze the data using Jumpshot, you first need to convert the AIX trace file, using the **uteconvert** utility, into UTE (Unified Trace Environment) interval files.

TYPE
```
uteconvert mytrace
```

where *mytrace* is the name of the trace file located in the current directory. *mytrace* is the prefix of the filename of the trace file. For example, if you had three tasks, the trace files would be named *mytrace0*, *mytrace1*, and *mytrace2*. This trace file has the same name as the file you specified for the output earlier in your example. This command will convert the trace file from AIX trace format into the UTE interval file.

Using the **-o** flag, you can optionally specify the name of the output UTE interval file. For example, to specify that the output file should be named *outputfile*,

TYPE
```
uteconvert -o outputfile mytrace
```

To convert a set of AIX trace files into a set of UTE interval files, specify the number of files using the –n option, and supply the common ″base name″ prefix shared by all of the files. For example, to convert five trace files with the prefix *mytraces* into UTE interval files, copy the trace files into a common directory,

TYPE
```
uteconvert -n 5 mytraces
```

Step 5 - Convert UTE interval files to SLOG2 files

First, review the differences between SLOG and SLOG2 files, to ensure that you are using the correct PE conversion utility. This is explained in “Step 1 - Determine which SLOG file to generate” on page 82.

Convert the UTE interval files into SLOG2 files using the **traceTOslog2** utility.

TYPE
```
traceTOslog2 mytrace.ute
```

where *mytrace* is the name of the UTE interval file.

The default output file name is the name of the input file, with **.slog2** appended. If more than one input file is processed, an output file name must be specified.

Use the **-o** option on the **traceTOslog2** command to specify an output file name. For example:

TYPE
```
traceTOslog2 -o mergedtrc.slog mytrace.ute
```

If you have multiple interval files, use **–n** to specify the number of files.

Note: If the traces were generated on a system without access to a switch, the **-g** flag is required when processing more than one input file.

Step 6 - Run Jumpshot

First, review the differences between SLOG and SLOG2 files, to ensure that you are using the correct PE conversion utility. This is explained in “Step 1 - Determine which SLOG file to generate” on page 82. Jumpshot is a public domain tool developed by Argonne National Laboratory and is not part of the PE Benchmarker Toolset.

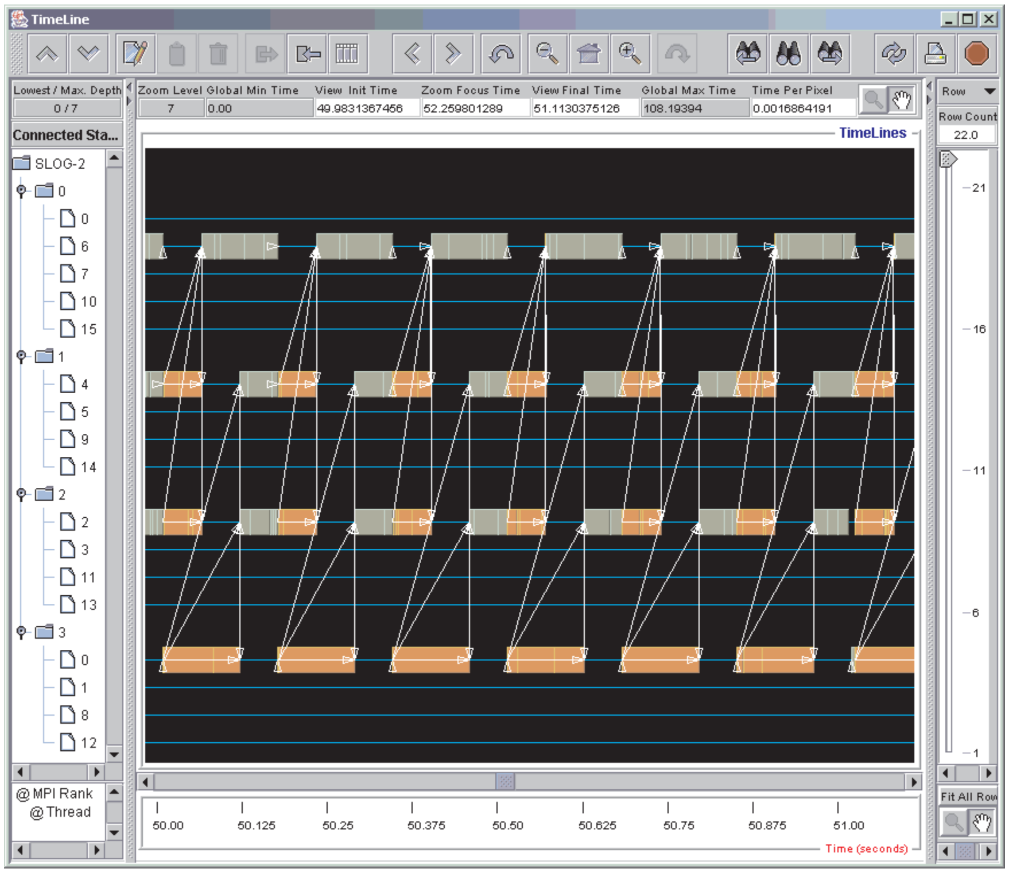
TYPE **jumpshot** to display the Jumpshot graphical user interface. (You have already downloaded the Jumpshot program available from Argonne National Laboratory).

SELECT
File → **Select** from the menu bar to load the SLOG2 file. Then select the SLOG2 file using the file selector dialog.

The window that appears displays the events of the program across a time line. To see detailed load balancing information, continue on with the next step.

CLICK
the **Display** button.

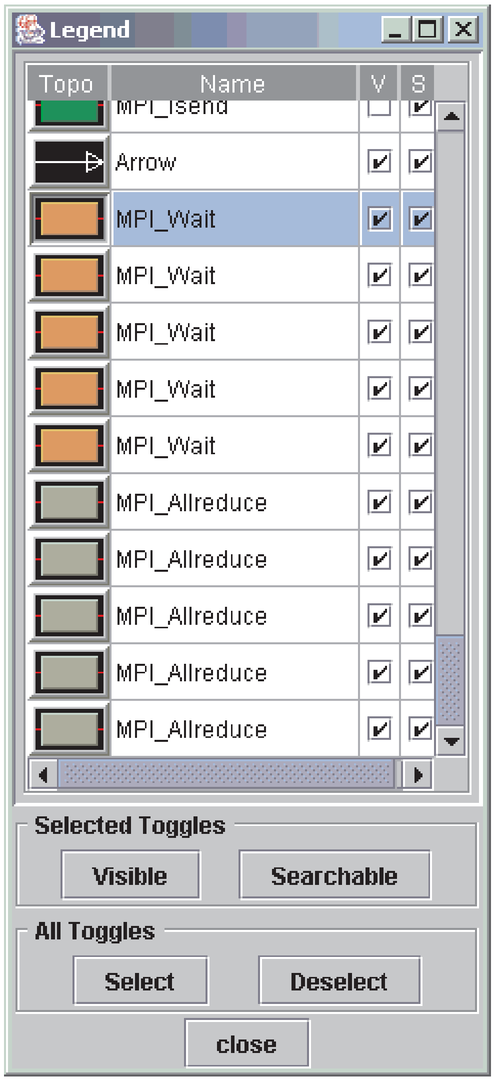
Figure 11 on page 86 illustrates the MPI functions occurring during the execution of the skewed program. Each box shown represents an MPI function and the arrows and lines represent communications calls between or within the functions.



Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel	Row Count
0 / 7	7	0.00	49.9831367456	52.259801289	51.1130375126	108.19394	0.0016864191	22.0

Figure 11. Jumpshot - skewed program

Figure 12 on page 87 shows the colors used to draw each interval. It also allows classes of intervals to be selected for display or searching, and to modify the colors of the intervals while viewing.



Topo	Name	V	S
🟩	MPI_Isend		✓
→	Arrow	✓	✓
🟧	MPI_Wait	✓	✓
🟧	MPI_Wait	✓	✓
🟧	MPI_Wait	✓	✓
🟧	MPI_Wait	✓	✓
🟧	MPI_Wait	✓	✓
▨	MPI_Allreduce	✓	✓
▨	MPI_Allreduce	✓	✓
▨	MPI_Allreduce	✓	✓
▨	MPI_Allreduce	✓	✓
▨	MPI_Allreduce	✓	✓

Figure 12. Jumpshot legend - skewed program

Step 7 - Analyze results, make changes, verify improvements

By looking at the message passing, you can see some peculiar characteristics of your program. For instance, you notice that many of the processors waste time by waiting for others to complete before they continue. These kinds of characteristics lead us to the conclusion that you have introduced very poor load balancing across tasks.

One way to alleviate this problem is to allow some processors to work ahead if they can deduce that another iteration will be necessary to find a solution. If a task's individual max is large enough on one iteration to force the global max to reiterate across the entire array, that task may continue on the next iteration when its west and north boundaries are received.

To illustrate this, use the pipelined.f program.

```
      iter_count = 0
      local_err = close_enough + 1
 100  CONTINUE
         iter_count = iter_count + 1
```

```
    CALL exch_in(stencil, m, n, local_err, global_err,
1      iter_count, close_enough)

    IF (MAX(global_err,local_err).GE.close_enough) THEN
       local_err = 0.0
       DO j=1, n-2
          DO i=1, m-2
             old_val = stencil(i,j)

             stencil(i,j) = ( stencil( i-1, j ) +
1                             stencil( i+1, j ) +
2                             stencil( i ,j-1) +
3                             stencil( i ,j+1) ) / 4

             local_err = MAX(local_err, ABS(old_val-stencil(i,j)))
          END DO
       END DO
    END IF

    CALL exch_out(stencil, m, n, global_err, local_err)

    IF(MOD(iter_count,100).EQ.0)PRINT *, iter_count, global_err
IF (MAX(global_err,local_err).GE.close_enough) GOTO 100
PRINT *, "convergence reached after", iter_count, "iterations."
```

As you can see on the following line:

```
IF(MAX(global_err,local_err).GE.close_enough) THEN
```

the program checks to see if the value of **local_err** is enough to allow this task to continue on the next iteration. These improvements to your program should result in improvement in your load balance as well.

Now, let's run your new code to see how this new version fares.

```
$ time poe pipelined
```

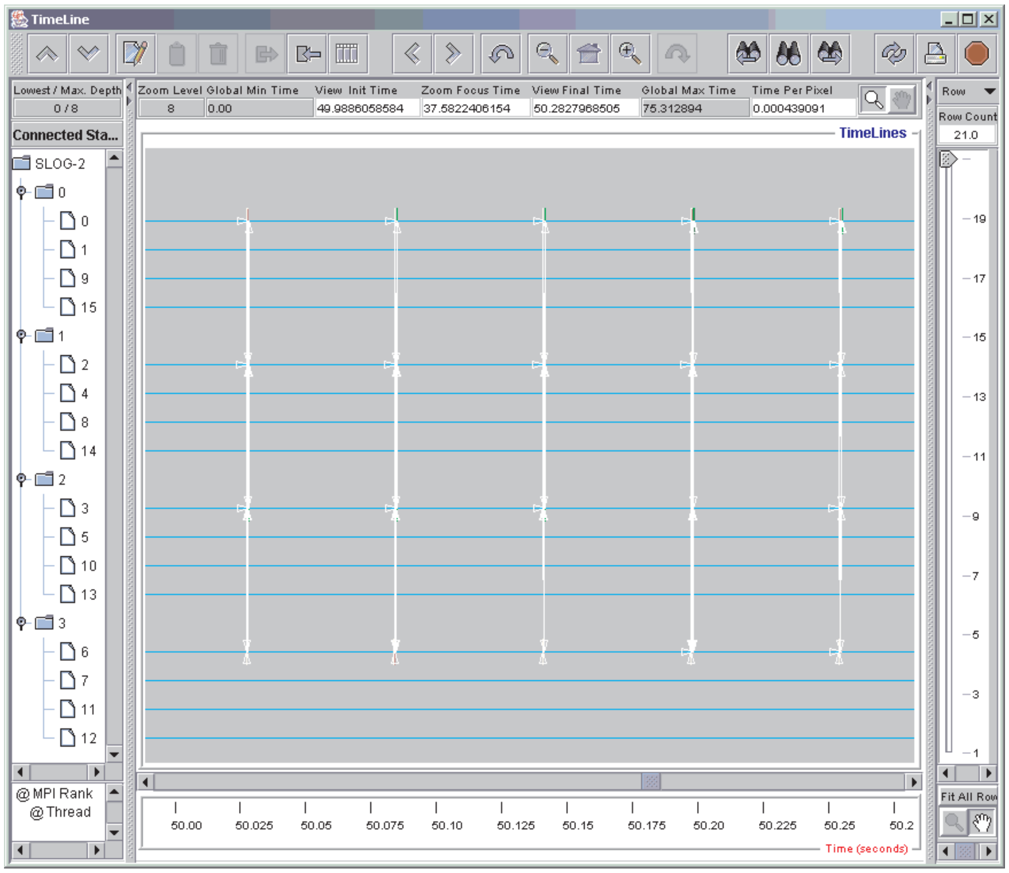
Table 6. Comparison of programs 'naive', 'reordered', 'chaotic', 'skewed', and 'pipelined'

Program Name	Tasks	Wallclock Time	Array Size per Task
naive	1 (single processor)	11 min. 1.94 sec.	1000 by 1000
reordered	1 (single processor)	5 min. 35.38 sec.	1000 by 1000
chaotic	4 (processors)	2 min. 4.58 sec.	500 by 500
skewed	4 (processors)	4 min. 41.87 sec.	500 by 500
pipelined	4 (processors)	2 min. 7.42 sec.	500 by 500

The execution times appearing in Table 6 were obtained with an earlier version of PE MPI on an IBM eServer clustered 1600 server. The execution times may be different, depending on the system that you are using, but the concepts for improving an application to reduce execution time are unchanged.

You were able to significantly improve the performance of your program and, at the same time, get a consistent, correct answer.

You can further analyze the pipelined program's load balance using Jumpshot. Figure 13 on page 89 illustrates that the load balance has improved in the pipelined program. This picture shows the communication patterns, but the interval between communications is so large that no detail can be seen in any sequence.



Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel	Row	Row Count
0 / 8	8	0.00	49.9886058584	37.5822406154	50.2827968505	75.312894	0.000439091		21.0

Figure 13. Jumpshot - pipelined program showing improved load balance

Figure 14 on page 90 is a closer look at a single communication sequence to see the detail of that sequence.

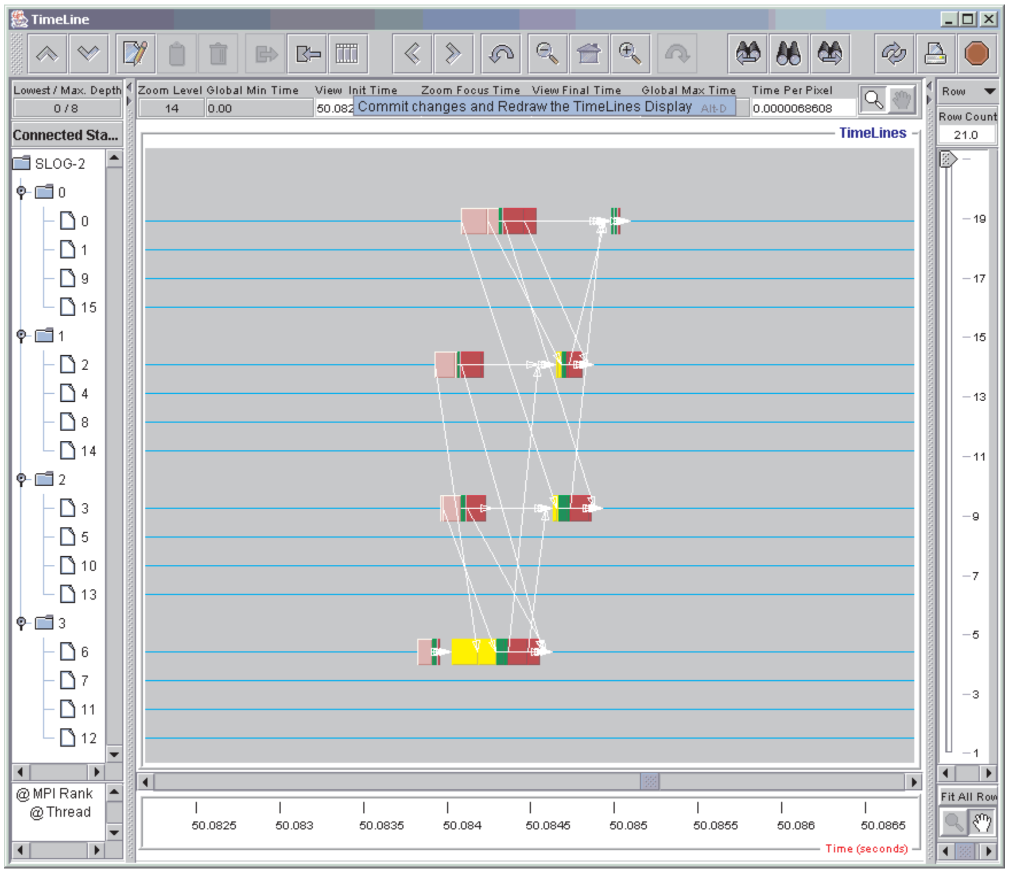
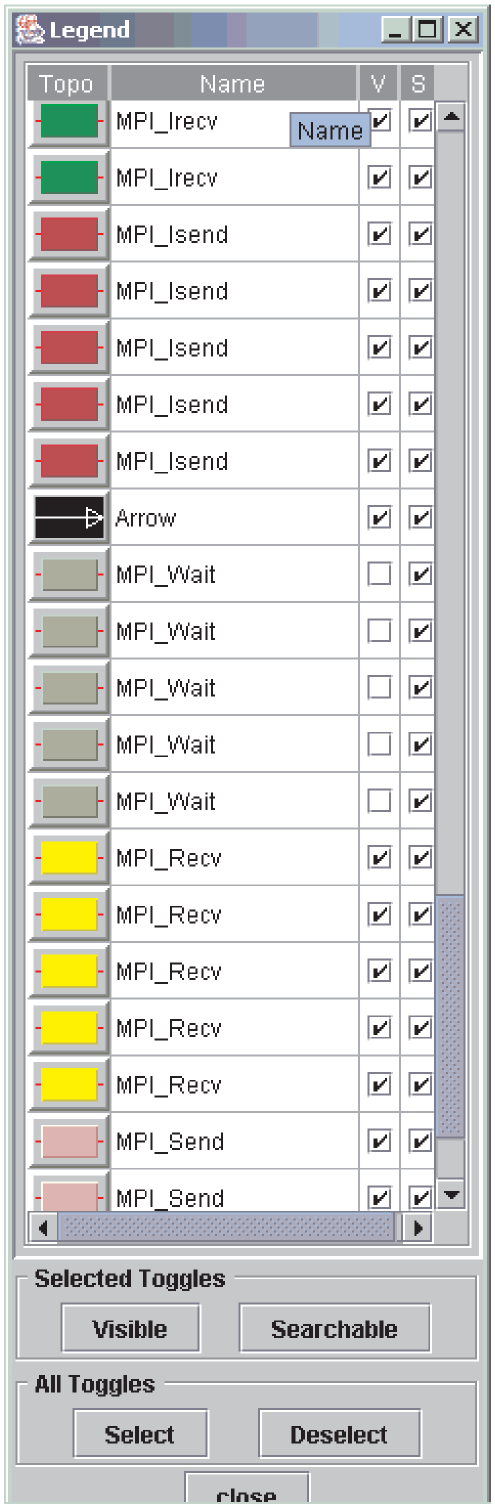


Figure 14. Jumpshot - pipielined program communication sequence

Figure 15 on page 91 shows the colors for the intervals in the other figures. It also allows classes of intervals to be selected for display or searching, and to modify the colors of the intervals while viewing.



Topo	Name	V	S
🟩	MPI_Irecv	✓	✓
🟩	MPI_Irecv	✓	✓
🟥	MPI_Isend	✓	✓
🟥	MPI_Isend	✓	✓
🟥	MPI_Isend	✓	✓
🟥	MPI_Isend	✓	✓
🟥	MPI_Isend	✓	✓
→	Arrow	✓	✓
⬜	MPI_Wait	☐	✓
⬜	MPI_Wait	☐	✓
⬜	MPI_Wait	☐	✓
⬜	MPI_Wait	☐	✓
⬜	MPI_Wait	☐	✓
🟨	MPI_Recv	✓	✓
🟨	MPI_Recv	✓	✓
🟨	MPI_Recv	✓	✓
🟨	MPI_Recv	✓	✓
🟨	MPI_Recv	✓	✓
🟥	MPI_Send	✓	✓
🟥	MPI_Send	✓	✓

Figure 15. Jumpshot legend – pipelined program

Tuning summary

Tuning the performance of a parallel application is no easier than tuning the performance of a sequential application. If anything, the parallel nature introduces another factor into the tuning equation. The approach PE has taken toward performance tuning is to provide tools which give you the information necessary to perform the tuning.

Chapter 5. Creating a safe program

Going from serial to parallel programming means that you are on a different scale now. There are some things that you need to pay attention to as you create your parallel programs. In particular, you need information on creating a *safe* MPI program. *MPI: A Message-Passing Interface Standard, Version 1.1*, which is available from the University of Tennessee (http://www.mpi-forum.org/) provides additional. information. You may want to refer to that document.

What is a safe program?

The MPI standard defines a program to as *safe* if message buffering is not required for the program to complete. In a program like this, you should be able to replace all standard sends with synchronous sends, and the program will still run correctly. This type of programming style is cleaner and more efficient; it provides good portability because program completion does not depend on the amount of available buffer space.

With PE, setting the **MP_EAGER_LIMIT** environment variable to **0** is equivalent to making all sends synchronous, including those used in collective communication. A good test of your program's safety is to set the **MP_EAGER_LIMIT** to **0**.

Some programmers prefer more flexibility and use an *unsafe* style that relies on buffering. This style is not recommended and you use it at your own risk. PE MPI does provide some buffer space to allow small messages to be processed efficiently. This buffer memory is not intended as a guarantee that an unsafe program will work. In many cases, this buffer space will allow an unsafe program to run but there is no assurance that it will still run with more tasks, different input data or on another implementation of MPI. You can use the buffered send mode for programs that require more buffering, or in situations where you want more control. Since buffer overflow conditions are easier to diagnose than deadlocks, you can also use this mode for debugging purposes.

You can use nonblocking message passing operations to avoid the need for buffering outgoing messages. This prevents deadlock situations due to a lack of buffer space, and improves performance by allowing computation and communication to overlap. It also avoids the overhead associated with allocating buffers and copying messages into buffers.

Safety and threaded programs

Sometimes message passing programs can hang or deadlock. This can occur when one task waits for a message that is never sent or when each task is waiting for the other task to send or receive a message. Within a task, a similar situation can occur when one thread is waiting for another thread to release a lock on a shared resource, such as a piece of memory. If thread *A*, which holds the lock, cannot run to the point at which it is ready to release it, the waiting thread *B* will never run. This may occur because thread *B* holds some other lock that thread *A* needs. Thread *A* cannot proceed until thread *B* does, and thread *B* cannot proceed until thread *A* does.

When programs are both multi-thread and multi-task, there is risk of *deadly embrace* involving both mutex and communication blocks. Say threads *A* and *B* are on task 0, and thread *A* holds a lock while waiting for a message from task 1.

Thread *B* will send a message to task 1 only after it gets the lock that thread *A* holds. If task 1 will send the message that thread *A* is waiting for only after getting the one that thread *B* cannot send, the job is in a 3-way deadly embrace, involving two threads at task 0 and one thread at task 1.

A problem that is more subtle occurs when two threads simultaneously access a shared resource without a lock protocol. The result may be incorrect without any obvious sign. For example, the following function is not threadsafe, because the thread may be preempted after the variable *c* is updated, but before it is stored.

```
int c;  /* external, used by two threads */
void update_it()
 {
     c++;  /* this is not threadsafe */
 {
```

You probably should avoid writing threaded message passing programs until you are familiar with writing and debugging threaded, single-task programs.

Using threaded programs with non-threadsafe libraries

A threaded MPI program must meet the same criteria as any other threaded program; it must avoid using non-threadsafe functions in more than one thread (for example, **strtok**). In addition, it must use only threadsafe libraries, if library functions are called on more than one thread. All of the libraries may not be threadsafe, so you should carefully examine how they are used in your program.

Message ordering

With MPI, messages are *non-overtaking*. This means that the order of sends must match the order of receives. Assume a sender sends two messages (Message 1 and Message 2) in succession, to the same destination, and both match the same receive. The receive operation will receive Message 1 before Message 2. Likewise, if a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2. Adhering to this rule ensures that sends are always matched with receives.

If a process in your program has a single thread of execution, then the sends and receives that occur follow a natural order. However, if a process has multiple threads, the various threads may not execute their relative send operations in any defined order. In this case, the messages can be received in any order.

Order rules apply within each communicator. Weakly synchronized threads can each use independent communicators to avoid many order problems.

The following is an example of using non-overtaking messages. The message sent by the first send must be received by the first receive, and the message sent by the second send must be received by the second receive.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE    ! rank.EQ.1
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

Program progress when two processes initiate two matching sends and receives

If two processes (or *tasks*) initiate two matching sends and receives, at least one of the operations (the send or the receive) will complete, regardless of other actions that occur in the system. The send operation will complete unless its matching receive operation has already been satisfied by another message, and has itself completed. Likewise, the receive operation will complete unless its matching send message is claimed by another matching receive that was posted at the same destination.

The following example shows two matching pairs that are intertwined in this manner. Here is what happens:

1. Both processes invoke their first calls.
2. *process 0*'s first send indicates buffered mode, which means it must complete, even if there is no matching receive. Since the first receive posted by *process 1* does not match, the send message gets copied into buffer space.
3. Next, *process 0* posts its second send operation, which matches *process 1*'s first receive, and both operations complete.
4. *process 1* then posts its second receive, which matches the buffered message, so both complete.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
    CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank.EQ.1
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

Communication fairness

MPI does not guarantee *fairness* in the way communications are handled. It is your responsibility to prevent starvation among the operations in your program.

One example of an *unfair* situation might be where a send, with a matching receive on another process, does not complete because another message, from a different process, overtakes the receive.

Resource limitations

If a lack of resources prevents an MPI call from executing, errors may result. Pending send and receive operations consume a portion of your system resources. MPI attempts to use a minimal amount of resource for each pending send and receive, but buffer space is required for storing messages sent in either standard or buffered mode when no matching receive is available.

When a buffered send operation cannot complete due to a lack of buffer space, the resulting error could cause your program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of a lack of buffer space, will block and wait for buffer space to become available or for the matching receive to be posted. In some situations, this behavior is preferable because it avoids the error condition associated with buffer overflow.

Sometimes a lack of buffer space can lead to deadlock. The program in the following example will succeed even if no buffer space for data is available.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE    ! rank.EQ.1
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

In this next example, neither process will send until the other process sends first. As a result, this program will always result in deadlock.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE    ! rank.EQ.1
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

The example below shows how message exchange relies on buffer space. The message send by each process must be copied out before the send returns and the receive starts. Consequently, at least one of the two messages sent needs to be buffered for the program to complete. As a result, this program can execute successfully only if the communication system can buffer at least the words of data specified by *count*.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE    ! rank.EQ.1
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

When standard send operations are used, deadlock can occur where both processes are blocked because buffer space is not available. This is also true for synchronous send operations. For buffered sends, if the required amount of buffer space is not available, the program will not complete either, and instead of deadlock, you will have buffer overflow.

Appendix A. A sample program to illustrate messages

This is sample output for a program run under POE with the maximum level of message reporting. There are also illustrations of the different types of messages you can expect, and their meaning.

To set the level of messages that are reported when you run your program, you can use the **-infolevel** (or **-ilevel**) option when you invoke POE. You can also use the **MP_INFOLEVEL** environment variable. Setting either of these to 6 gives you the maximum number of diagnostic messages when you run your program. For more information about setting the POE message level, see *IBM Parallel Environment: Operation and Use, Volume 1*.

Note that we are using numbered prefixes along the left-hand edge of the following output as a way to refer to particular lines. The prefixes are **not** part of the output you will see when you run your program. For an explanation of the messages denoted by these numbered prefixes, see “Figuring out what all of this means” on page 99.

This command produces output similar to the following:

```
> poe hello_world_c -procs 2  -rmpool 1 -infolevel 6
```

```
    +1  INFO: DEBUG_LEVEL changed from 0 to 4
    +2  D1<L4>: Open of file ./host.list successful
    +3  ATTENTION: 0031-379  Pool setting ignored when
        hostfile used
    +4  D1<L4>: mp_euilib = ip
    +5  D1<L4>: 03/04 13:55:37.682266  task 0
        k151f1rp02.kgn.ibm.com 89.116.177.5 11
    +6  D1<L4>: 03/04 13:55:37.684025  task 1
        k151f1rp02.kgn.ibm.com 89.116.177.5 11
    +7  D1<L4>: node allocation strategy = 0
    +8  D1<L4>: Entering pm_contact, jobid is 0
    +9  D1<L4>: Jobid = 1110376467
   +10  D1<L4>: POE security method is COMPAT
   +11  D1<L4>: Requesting service pmv4
   +12  D1<L4>: 1 master nodes
   +13  D4<L4>: LoadLeveler Version 0 Release 0
   +14  D1<L4>: Socket file descriptor for master 0
        (k151f1rp02.kgn.ibm.com) is 4
   +15  D1<L4>: SSM_read on socket 4, source = 0,
        task id: 0, nread: 12, type:3.
   +16  D1<L4>: Leaving pm_contact, jobid
        is 1110376467
   +17  D1<L4>: attempting to bind socket
        to /tmp/s.pedb.413930.1079
   +18
   +19  D4<L4>: Command args:<>
   +20  D3<L4>: Message type 34 from source 0
   +21  D4<L4>: Task 0 pulse received,count is 0
        curr_time is 1109962537
   +22  D4<L4>: Task 0 pulse acknowledged, count is 0
        curr_time is 1109962537
   +23  D3<L4>: Message type 21 from source 0
   +24  INFO: 0031-724  Executing program:
        <../../hello_world_c>
   +25  D3<L4>: Message type 21 from source 0
   +26  D1<L4>: Affinity is not requested;
        MP_TASK_AFFINITY: -1
   +27  D3<L4>: Message type 21 from source 1
   +28  D3<L4>: Message type 21 from source 1
   +29  INFO: 0031-724  Executing program:
        <../../hello_world_c>
   +30  D1<L4>: Affinity is not requested;
        MP_TASK_AFFINITY: -1
   +31  D3<L4>: Message type 21 from source 0
   +32  INFO: DEBUG_LEVEL changed from 0 to 4
   +33  D3<L4>: Message type 21 from source 0
   +34  D3<L4>: Message type 21 from source 0
   +35  D4<L4>: pm_async_thread sends cond sig
   +36  D4<L4>: pm_async_thread calls sigwait,
         in_async_thread=0
   +37  D4<L4>: pm_main, wake up from timed cond wait
   +38  D1<L4>: In mp_main, mp_main will not
        be checkpointable
   +39  D3<L4>: Message type 21 from source 0
   +40  D1<L4>: mp_euilib is <ip>
   +41  D3<L4>: Message type 21 from source 0
   +42  D1<L4>: Executing _mp_init_msg_passing()
        from MPI_Init()...
   +43  D3<L4>: Message type 21 from source 0
   +44  D1<L4>: mp_css_interrupt is <0>
   +45  D1<L4>: About to call mpci_connect
   +46  D3<L4>: Message type 21 from source 1
   +47  INFO: DEBUG_LEVEL changed from 0 to 4
   +48  D3<L4>: Message type 21 from source 1
   +49  D4<L4>: pm_async_thread sends cond sig
   +50  D3<L4>: Message type 21 from source 0
   +51  INFO: 0031-619  32bit(ip)
        MPCI shared object was compiled at
        Wed Mar  2 13:44:02 2005
   +52
   +53  D3<L4>: Message type 21 from source 1
   +54  D4<L4>: pm_async_thread calls sigwait, in_async_thread=0
   +55  D4<L4>: pm_main, wake up from timed cond wait
   +56  D1<L4>: In mp_main, mp_main will not be checkpointable
   +57  D1<L4>: mp_euilib is <ip>
   +58  D3<L4>: Message type 21 from source 1
   +59  D1<L4>: Executing _mp_init_msg_passing() from MPI_Init()...
   +60  D3<L4>: Message type 21 from source 1
   +61  D1<L4>: mp_css_interrupt is <0>
   +62  D1<L4>: About to call mpci_connect
   +63  D3<L4>: Message type 21 from source 0
   +64  LAPI version #6.61 2005/01/28 1.143.1.3 src/rsct/lapi/lapi.c,
        lapi, rsct_rag2, rag20508a 32bit(ip)  library compiled on
        Wed Mar  2 11:46:57 2005
   +65  .
   +66  D3<L4>: Message type 21 from source 0
   +67  LAPI is using lightweight lock.
   +68  D3<L4>: Message type 21 from source 1
   +69  LAPI version #6.61 2005/01/28 1.143.1.3 src/rsct/lapi/lapi.c, lapi,
         rsct_rag2, rag20508a 32bit(ip)  library compiled on
         Wed Mar  2 11:46:57 2005
   +70  .
   +71  D3<L4>: Message type 21 from source 1
   +72  LAPI is using lightweight lock.
   +73  D3<L4>: Message type 23 from source 0
   +74  D1<L4>: init_data for instance number 0,
        task 0: <158498562:37292>
   +75  D3<L4>: Message type 23 from source 1
   +76  D1<L4>: init_data for instance number 0,
        task 1: <158498562:37293>
   +77  D3<L4>: Message type 21 from source 1
   +78  The MPI shared memory protocol is used for the job
   +79  D3<L4>: Message type 21 from source 0
   +80  The MPI shared memory protocol is used for the job
   +81  D1<L4>: Elapsed time for mpci_connect: 1 seconds
   +82  D3<L4>: Message type 21 from source 1
   +83  D1<L4>: Elapsed time for mpci_connect: 1 seconds
```

```
 +84  D3<L4>: Message type 21 from source 0
 +85  D1<L4>: _css_init: rc from HPSOclk_init is 1
 +86
 +87  D1<L4>: About to call _ccl_init
 +88  D3<L4>: Message type 21 from source 1
 +89  D1<L4>: _css_init: rc from HPSOclk_init is 1
 +90
 +91  D1<L4>: About to call _ccl_init
 +92  D3<L4>: Message type 88 from source 0
 +93  D3<L4>: Message type 88 from source 1
 +94  D3<L4>: Message type 21 from source 0
 +95  D2<L4>: Global Data for
      task 0: 1;0,89.116.177.5,-3;778658413,89.116.177.5,-3;
 +96  D3<L4>: Message type 21 from source 1
 +97  D2<L4>: Global Data for
      task 1: 1;0,89.116.177.5,-3;778658413,89.116.177.5,-3;
 +98  D3<L4>: Message type 21 from source 0
 +99  D1<L4>: Elapsed time for _ccl_init: 0 seconds
+100  D3<L4>: Message type 21 from source 1
+101  D1<L4>: Elapsed time for _ccl_init: 0 seconds
+102  D3<L4>: Message type 20 from source 0
+103  Hello World !!
+104  D3<L4>: Message type 62 from source 0
+105  D3<L4>: Message type 20 from source 1
+106  Hello World !!
+107  D3<L4>: Message type 62 from source 1
+108  D3<L4>: Message type 21 from source 0
+109  INFO: 0031-306  pm_atexit: pm_exit_value is 0.
+110  D3<L4>: Message type 17 from source 0
+111  D3<L4>: Message type 21 from source 1
+112  INFO: 0031-306  pm_atexit: pm_exit_value is 0.
+113  D3<L4>: Message type 17 from source 1
+114  D3<L4>: Message type 22 from source 0
+115  INFO: 0031-656  I/O file STDOUT closed by task 0
+116  D3<L4>: Message type 22 from source 0
+117  INFO: 0031-656  I/O file STDERR closed by task 0
+118  D3<L4>: Message type 22 from source 1
+119  INFO: 0031-656  I/O file STDOUT closed by task 1
+120  D3<L4>: Message type 22 from source 1
+121  INFO: 0031-656  I/O file STDERR closed by task 1
+122  D3<L4>: Message type 15 from source 0
+123  D1<L4>: Accounting data from task 0 for source 0:
+124  D3<L4>: Message type 15 from source 1
+125  D1<L4>: Accounting data from task 1 for source 1:
+126  D3<L4>: Message type 1 from source 0
+127  INFO: 0031-251  task 0 exited: rc=0
+128  D3<L4>: Message type 1 from source 1
+129  INFO: 0031-251  task 1 exited: rc=0
+130  D1<L4>: All remote tasks have exited: maxx_errcode = 0
+131  INFO: 0031-639  Exit status from pm_respond = 0
+132  D1<L4>: Maximum return code from user = 0
+133  D2<L4>: In pm_exit... About to call pm_remote_shutdown
+134  D2<L4>: Sending PMD_EXIT to task 0
+135  D2<L4>: Elapsed time for pm_remote_shutdown: 0 seconds
+136  D2<L4>: In pm_exit... Calling exit with status = 0 at
       Fri Mar  4 13:55:38 2005
```

Figuring out what all of this means

When you set **-infolevel** to 6, you get the full complement of diagnostic messages.

The example in Appendix A, “A sample program to illustrate messages,” on page 97 includes numbered prefixes along the left-hand edge of the output so that you can refer to particular lines, and then explain what they mean. Remember, that these prefixes are **not** part of your output. This list points you to the line number of the messages that are of most interest, and provides a short description of each.

Line number	Message description
5-6	Names hosts that are used.
10	Indicates security method defined on the remote node.
11	Indicates that service **pmv4**, from **/etc/services** is being used.
14	Indicates node with partition manager running.
20	Message type 34 indicates pulse activity (the pulse mechanism checked that each remote node was actively participating with the home node).
23	Message type 21 indicates a STDERR message.
40, 57	Indicates that the euilib message passing protocol was specified.
42, 59	Indicates that message passing initialization has begun.
51	Timestamp of MPCI shared object being executed.
64, 69	Timestamp of LAPI library being executed.
78, 80	Indicates that MPI shared memory is being used.
81, 83	Indicates that initialization of MPCI has completed.
92, 93, 95, 97	Message type 88 shows MPI global task information.
102, 103, 105, 106	Message type 20 shows STDOUT from your program.
109, 112	Indicates that the user's program has reached the exit handler. The exit code is 0.
110, 113	Message type 17 indicates the tasks have requested to exit.
115, 117, 119, 121	Indicates that the STDOUT and STDERR pipes have been closed.
122, 124	Message type 15 indicates accounting data.
134	Indicates that the home node is sending an exit.

Appendix B. Parallel Environment internals

This is some additional information about how the IBM Parallel Environment (PE) works with respect to your application. Much of this information is also explained in the *IBM Parallel Environment: MPI Programming Guide*.

What happens when I compile my applications?

In order to run your program in parallel, you first need to compile your application source code with one of the following scripts:

1. **mpcc_r**
2. **mpCC_r**
3. **mpxlf_r**
4. **mpxlf95_r**
5. **mpxlf90_r**

To make sure the parallel execution works, these scripts add the following to your application executable:

- POE initialization module, so POE can determine that all nodes can communicate successfully, before giving control to the user application's main() routine.
- Signal handlers, for additional control in terminating the program during parallel tracing, and enabling the handling of the process termination signals. The *IBM Parallel Environment: MPI Programming Guide* explains the signals that are handled in this manner.

The compile scripts dynamically link the Message Passing library interfaces in such a way that the specific communication library that is used is determined when your application executes.

Applications created as static executables are not supported.

How do my applications start?

Because POE adds its entry point to each application executable, user applications do not need to be run under the **poe** command. When a parallel application is invoked directly, as opposed to under the control of the **poe** command, POE is started automatically. It then sets up the parallel execution environment and then re-invokes the application on each of the remote nodes.

Serial applications can be run in parallel only using the **poe** command. However, such applications cannot take advantage of the function and performance provided with the message passing libraries.

How does POE talk to the nodes?

A parallel job running under POE consists of a *home node* (where POE was started) and *n* tasks. Each task runs under the control of a Partition Manager daemon (pmd). There is one pmd for each job on each node on which the job's tasks run.

When you start a parallel job, POE contacts the nodes assigned to run the job (called *remote nodes*), and starts a pmd instance on each node. POE sends environment information to the pmd daemons for the parallel job (including the name of the executable) and the pmd daemons spawn processes to run the executable. For tasks that run on the same node, the pmd daemon forks and manages all tasks for that job on that node. It routes messages to and from each remote task, and also coordinates with the home node to terminate each task.

The spawned processes have standard I/O redirected to socket connections back to the pmd daemons. Therefore, any output the application writes to STDOUT or STDERR is sent back to the pmd daemons. The pmd daemons, in turn, send the output back to POE via another socket connection, and POE writes the output to its STDOUT or STDERR. Any input that POE receives on STDIN is delivered to the remote tasks in a similar fashion.

The socket connections between POE and the pmd daemons are also used to exchange control messages for providing task synchronization, exit status, and signaling. These capabilities are available to control any parallel program run by POE, and they do not depend on the message passing library.

When POE executes without LoadLeveler, it is assumed that the Partition Manager Daemon (PMD) is started under **inetd**. There is no consideration for running the PMD without **inetd**.

When POE executes under LoadLeveler (including all User Space applications), the PMD is started by LoadLeveler.

How are signals handled?

POE installs signal handlers for most signals that cause program termination and interrupts, in order to control and notify all tasks of the signal. POE will exit the program normally with a code of (128 + signal). If the user program installs a signal handler for any of the signals POE supports, it should follow the guidelines presented in *IBM Parallel Environment: MPI Programming Guide*.

What happens when my application ends?

POE returns exit status (a return code value between 0 and 255) on the home node which reflects the composite exit status of the user application. The exit status can have various conditions and values and each can have a specific meaning. These are explained in The *IBM Parallel Environment: MPI Programming Guide*.

In addition, if the POE job-step function is used, the job control mechanism is the program's exit code. When the task exit code is 0 (zero), or in the range of 2 to 127, the job-step will be continued. If the task exit code is 1 or greater than 127, POE terminates the parallel job, as well as any remaining user programs in the job-step list. Also, any POE infrastructure failure detected (such as failure to open pipes to the child process) will terminate the parallel job as well as any remaining programs in the job-step list.

Appendix C. Accessibility features for PE

Accessibility features help a user who has a physical 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 IBM Parallel Environment. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- Keys that are tactilely discernible and do not activate just by touching them.
- Industry-standard devices for ports and connectors.
- The attachment of alternative input and output devices.

Note: The IBM eServer Cluster Information Center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard navigation

This product uses standard Microsoft® Windows® navigation keys.

IBM and accessibility

See the *IBM Accessibility Center* at **http://www.ibm.com/able** for more information about the commitment that IBM has to accessibility.

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 (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.

This book refers to IBM's implementation of the Message Passing Interface (MPI) standard for Parallel Environment for AIX (PE). PE MPI intends to comply with the requirements of the Message Passing Interface Forum described below. PE MPI provides an implementation of MPI which is complete except for omitting the features described in the ″Process Creation and Management″ chapter of MPI-2.

Permission to copy without fee all or part of these Message Passing Interface Forum documents:

Trademarks

Acknowledgements

The PE Benchmarker product includes software developed by the Apache Software Foundation, *http://www.apache.org*.

Glossary

A

AFS. Andrew File System.

address. A value, possibly a character or group of characters that identifies a register, a device, a particular part of storage, or some other data source or destination.

AIX. Abbreviation for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system. AIX is particularly suited to support technical computing applications, including high-function graphics and floating-point computations.

API. Application programming interface.

application. The use to which a data processing system is put; for example, a payroll application, an airline reservation application.

argument. A parameter passed between a calling program and a called program or subprogram.

attribute. A named property of an entity.

Authentication. The process of validating the identity of a user or server.

Authorization. The process of obtaining permission to perform specific actions.

B

bandwidth. For a specific amount of time, the amount of data that can be transmitted. Bandwidth is expressed in bits or bytes per second (bps) for digital devices, and in cycles per second (Hz) for analog devices.

blocking operation. An operation that does not complete 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 operation. A communication operation where one processor sends (or broadcasts) a message to all other processors.

buffer. A portion of storage used to hold input or output data temporarily.

C

C. A general-purpose programming language. It was formalized by Uniforum in 1983 and the ANSI standards committee for the C language in 1984.

C++. A general-purpose programming language that is based on the C language. C++ includes extensions that support an object-oriented programming paradigm. Extensions include:
- strong typing
- data abstraction and encapsulation
- polymorphism through function overloading and templates
- class inheritance.

chaotic relaxation. An iterative relaxation method that uses a combination of the Gauss-Seidel and Jacobi-Seidel methods. The array of discrete values is divided into subregions that can be operated on in parallel. The subregion boundaries are calculated using the Jacobi-Seidel method, while the subregion interiors are calculated using the Gauss-Seidel method. See also *Gauss-Seidel*.

client. A function that requests services from a server and makes them available to the user.

cluster. A group of processors interconnected through a high-speed network that can be used for high-performance computing.

Cluster 1600. See IBM eServer Cluster 1600.

collective communication. A communication operation that involves more than two processes or tasks. Broadcasts, reductions, and the **MPI_Allreduce** subroutine are all examples of collective communication operations. All tasks in a communicator must participate.

command alias. When using the PE command-line debugger **pdbx**, you can create abbreviations for existing commands using the **pdbx alias** command. These abbreviations are known as *command aliases.*

communicator. An MPI object that describes the communication context and an associated group of processes.

compile. To translate a source program into an executable program.

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. The current program state is needed for the programmer to diagnose and correct the problem.

core file. A file that preserves the state of a program, usually just before a program is terminated for an unexpected error. See also *core dump.*

current context. When using the **pdbx** debugger, control of the parallel program and the display of its data can be limited to a subset of the tasks belonging to that program. This subset of tasks is called the *current context.* You can set the current context to be a single task, multiple tasks, or all the tasks in the program.

D

data decomposition. A method of breaking up (or decomposing) a program into smaller parts to exploit parallelism. One divides the program by dividing the data (usually arrays) into smaller parts and operating on each part independently.

data parallelism. Refers to situations where parallel tasks perform the same computation on different sets of data.

dbx. A symbolic command-line debugger that is often provided with UNIX systems. The PE command-line debugger **pdbx** is based on the **dbx** debugger.

debugger. A debugger provides an environment in which you can manually control the execution of a program. It also provides the ability to display the program's data and operation.

distributed shell (dsh). An IBM AIX Parallel System Support Programs command that lets you issue commands to a group of hosts in parallel. See *IBM Parallel System Support Programs for AIX: Command and Technical Reference* for details.

domain name. The hierarchical identification of a host system (in a network), consisting of human-readable labels, separated by decimal points.

DPCL target application. The executable program that is instrumented by a Dynamic Probe Class Library (DPCL) analysis tool. It is the process (or processes) into which the DPCL analysis tool inserts probes. A target application could be a serial or parallel program. Furthermore, if the target application is a parallel program, it could follow either the SPMD or the MPMD model, and may be designed for either a message-passing or a shared-memory system.

E

environment variable. (1) A variable that describes the operating environment of the process. Common environment variables describe the home directory, command search path, and 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 baseband local area network (LAN) that allows multiple stations to access the transmission medium at will without prior coordination, avoids contention by using carrier sense and deference, and resolves contention by using collision detection and delayed retransmission. Ethernet uses carrier sense multiple access with collision detection (CSMA/CD).

event. An occurrence of significance to a task — the completion of an asynchronous operation such as an input/output operation, for example.

executable. A program that has been link-edited and therefore can be run in a processor.

execution. To perform the actions specified by a program or a portion of a program.

expression. In programming languages, a language construct for computing a value from one or more operands.

F

fairness. A policy in which tasks, threads, or processes must be allowed eventual 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 local area network (LAN) using optical fiber cables. An FDDI LAN can be up to 100 kilometers (62 miles) long, and can include up to 500 system units. There can be up to 2 kilometers (1.24 miles) between system units and concentrators.

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 functions. 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.*

foreign host. See *remote host.*

FORTRAN. One of the oldest of the modern programming languages, and the most popular language for scientific and engineering computations. Its name is a contraction of *FORmula TRANslation.* The two most common FORTRAN versions are FORTRAN

77, originally standardized in 1978, and FORTRAN 90. FORTRAN 77 is a proper subset of FORTRAN 90.

function cycle. A chain of calls in which the first caller is also the last to be called. A function that calls itself recursively is not considered a function cycle.

functional decomposition. A method of dividing the work in a program to exploit parallelism. The program is divided into independent pieces of functionality, which are distributed to independent processors. This method is in contrast to data decomposition, which distributes the same work over different data to independent processors.

functional parallelism. Refers to situations where parallel tasks specialize in particular work.

G

Gauss-Seidel. An iterative relaxation method for solving Laplace's equation. It calculates the general solution by finding particular solutions to a set of discrete points distributed throughout the area in question. The values of the individual points are obtained by averaging the values of nearby points. Gauss-Seidel differs from Jacobi-Seidel in that, for the *i+1*st iteration, Jacobi-Seidel uses only values calculated in the *i*th iteration. Gauss-Seidel uses a mixture of values calculated in the *i*th and *i+1*st iterations.

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 variable defined in one portion of a computer program and used in at least one other portion of the computer program.

gprof. A UNIX command that produces an execution profile of C, COBOL, FORTRAN, or Pascal programs. The execution profile is in a textual and tabular format. It is useful for identifying which routines use the most CPU time. See the man page on **gprof**.

graphical user interface (GUI). A type of computer interface consisting of a visual metaphor of a real-world scene, often of a desktop. Within that scene are icons, which represent actual objects, that the user can access and manipulate with a pointing device.

GUI. Graphical user interface.

H

| **high performance switch.** The high-performance
| message-passing network that connects all processor
| nodes together.

hook. A **pdbx** command that lets you re-establish control over all tasks in the current context that were previously unhooked with this command.

home node. The node from which an application developer compiles and runs his program. The home node can be any workstation on the LAN.

host. A computer connected to a network that provides an access method to that network. A host provides end-user services.

host list file. A file that contains a list of host names, and possibly other information, that was defined by the application that reads it.

host name. The name used to uniquely identify any computer on a network.

hot spot. A memory location or synchronization resource for which multiple processors compete excessively. This competition can cause a disproportionately large performance degradation when one processor that seeks the resource blocks, preventing many other processors from having it, thereby forcing them to become idle.

I

| **IBM eServer Cluster 1600.** An IBM eServer Cluster
| 1600 is any CSM-managed cluster comprised of
| POWER™ microprocessor based systems (including
| RS/6000® SMPs, RS/6000 SP nodes, and pSeries
| SMPs).

IBM Parallel Environment (PE) for AIX. A licensed program that provides an execution and development environment for parallel C, C++, and FORTRAN programs. It also includes tools for debugging, profiling, and tuning parallel programs.

| **installation image.** A file or collection of files that are
| required in order to install a software product on system
| nodes. These files are in a form that allows them to be
| installed or removed with the AIX **installp** command.
| See also *fileset*, *licensed program*, and *package.*

Internet. The collection of worldwide networks and gateways that function as a single, cooperative virtual network.

| **Internet Protocol (IP).** The IP protocol lies beneath
| the UDP protocol, which provides packet delivery
| between user processes and the TCP protocol, which
| provides reliable message delivery between user
| processes.

IP. Internet Protocol.

J

Jacobi-Seidel. See *Gauss-Seidel.*

K

Kerberos. A publicly available security and authentication product that works with the IBM AIX Parallel System Support Programs software to authenticate the execution of remote commands.

kernel. The core portion of the UNIX operating system that controls the resources of the CPU and allocates them to the users. The kernel is memory-resident, is said to run in *kernel mode* (in other words, at higher execution priority level than *user mode*), and is protected from user tampering by the hardware.

L

Laplace's equation. A homogeneous partial differential equation used to describe heat transfer, electric fields, and many other applications.

latency. The time interval between the initiation of a send by an origin task and the completion of the matching receive by the target task. More generally, latency is the time between a task initiating data transfer and the time that transfer is recognized as complete at the data destination.

licensed program. A collection of software packages sold as a product that customers pay for to license. A licensed program can consist of packages and file sets a customer would install. These packages and file sets bear a copyright and are offered under the terms and conditions of a licensing agreement. See also *fileset* and *package.*

lightweight corefiles. An alternative to standard AIX corefiles. Corefiles 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 corefiles.

LoadLeveler. A job management system that works with POE to let users run jobs and match processing needs with system resources, in order to make better use of the system.

local variable. A variable that is defined and used only in one specified portion of a computer program.

loop unrolling. A program transformation that makes multiple copies of the body of a loop, also placing the copies within the body of the loop. The loop trip count and index are adjusted appropriately so the new loop computes the same values as the original. This transformation makes it possible for a compiler to take additional advantage of instruction pipelining, data cache effects, and software pipelining.

See also *optimization.*

M

management domain . A set of nodes configured for manageability by the Clusters Systems Management (CSM) product. Such a domain has a management server that is used to administer a number of managed nodes. Only management servers have knowledge of the whole domain. Managed nodes only know about the servers managing them; they know nothing of each other. Contrast with *peer domain*.

menu. A list of options displayed to the user by a data processing system, from which the user can select an action to be initiated.

message catalog. A file created from a message source file that contains application error and other messages, which can later be translated into other languages without having to recompile the application source code.

message passing. Refers to the process by which parallel tasks explicitly exchange program data.

Message Passing Interface (MPI). A standardized API for implementing the message-passing model.

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.

MPI. Message Passing Interface.

N

network. An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

Network Information Services. 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.** (1) In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network. (2) A single location or workstation in a network. Usually a physical entity, such as a processor.

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 was completed. For example, a nonblocking receive will not wait until a message arrives. By contrast, a blocking receive will wait. A nonblocking receive must be completed by a later test or wait.

O

object code. The result of translating a computer program to a relocatable, low-level form. Object code contains machine instructions, but symbol names (such as array, scalar, and procedure names), are not yet given a location in memory. Contrast with *source code.*

optimization. A widely-used (though not strictly accurate) term for program performance improvement, especially for performance improvement done by a compiler or other program translation software. An optimizing compiler is one that performs extensive code transformations in order to obtain an executable that runs faster but gives the same answer as the original. Such code transformations, however, can make code debugging and performance analysis very difficult because complex code transformations obscure the correspondence between compiled and original 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. A number of file sets that have been collected into a single installable image of licensed programs. Multiple file sets can be bundled together for installing groups of software together. See also *fileset* and *licensed program.*

parallelism. The degree to which parts of a program may be concurrently executed.

parallelize. To convert a serial program for parallel execution.

Parallel Operating Environment (POE). An execution environment that smooths the differences between serial and parallel execution. It lets you submit and manage parallel jobs. It is abbreviated and commonly known as POE.

parameter. (1) In FORTRAN, a symbol that is given a constant value for a specified application. (2) An item in a menu for which the operator specifies a value or for which the system provides a value when the menu is interpreted. (3) A name in a procedure that is used to refer to an argument that is passed to the procedure. (4) A particular piece of information that a system or application program needs to process a request.

| **partition.** (1) A fixed-size division of storage. (2) A
| logical collection of nodes to be viewed as one system
| or domain. System partitioning is a method of
| organizing the system into groups of nodes for testing
| or running different levels of software of product
| environments.

Partition Manager. The component of the Parallel Operating Environment (POE) that allocates nodes, sets up the execution environment for remote tasks, and manages distribution or collection of standard input (STDIN), standard output (STDOUT), and standard error (STDERR).

pdbx. The parallel, symbolic command-line debugging facility of PE. **pdbx** is based on the **dbx** debugger and has a similar interface.

PE. The Parallel Environment for AIX licensed program.

peer domain. A set of nodes configured for high availability by the RSCT configuration manager. 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*.

performance monitor. A utility that displays how effectively a system is being used by programs.

PID. Process identifier.

POE. Parallel Operating Environment.

| **pool.** Groups of nodes on a system that are known to
| LoadLeveler, and are identified by a pool name or
| number.

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. (1) In a programming language, a block, with or without formal parameters, whose execution is invoked 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. It 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.

prof. A utility that produces an execution profile of an application or program. It is useful to identify which routines use the most CPU time. See the man page for **prof**.

profiling. The act of determining how much CPU time is used by each function or subroutine in a program. The histogram or table produced is called the execution profile.

pthread. A thread that conforms to the POSIX Threads Programming Model.

R

reduced instruction-set computer. A computer that uses a small, simplified set of frequently-used instructions for rapid execution.

reduction operation. An operation, usually mathematical, that reduces a collection of data by one or more dimensions. For example, the arithmetic SUM operation is a reduction operation that reduces an array to a scalar value. Other reduction operations include MAXVAL and MINVAL.

Reliable Scalable Cluster Technology. A set of software components that together provide a comprehensive clustering environment for AIX. RSCT is the infrastructure used by a variety of IBM products to provide clusters with improved system availability, scalability, and ease of use.

remote host. Any host on a network except the one where a particular operator is working.

remote shell (rsh). A command that lets you issue commands on a remote host.

RISC. See *reduced instruction-set computer*.

RSCT. See *Reliable Scalable Cluster Technology*.

RSCT peer domain. See *peer domain*.

S

shell script. A sequence of commands that are to be executed by a shell interpreter such as the Bourne shell (**sh**), the C shell (**csh**), or the Korn shell (**ksh**). Script commands are stored in a file in the same format as if they were typed at a terminal.

segmentation fault. A system-detected error, usually caused by referencing an non-valid memory address.

server. A functional unit that provides shared services to workstations over a network — a file server, a print server, or a mail server, for example.

signal handling. In the context of a message passing library (such as MPI), there is a need for asynchronous operations to manage packet flow and data delivery while the application is doing computation. This asynchronous activity can be carried out either by a signal handler or by a service thread. The early IBM message passing libraries used a signal handler and the more recent libraries use service threads. The older libraries are often referred to as the *signal handling* versions.

Single program, multiple data (SPMD). A parallel programming model in which different processors execute the same program on different sets of data.

source code. The input to a compiler or assembler, written in a source language. Contrast with *object code.*

source line. A line of source code.

SPMD. Single program, multiple data.

standard error (STDERR). An output file intended to be used for error messages for C programs.

standard input (STDIN). The primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

standard output (STDOUT). The primary destination of data produced by a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command.

STDERR. Standard error.

STDIN. Standard input.

STDOUT. Standard output.

stencil. A pattern of memory references used for averaging. A 4-point stencil in two dimensions for a given array cell, x(i,j), uses the four adjacent cells, x(i-1,j), x(i+1,j), x(i,j-1), and x(i,j+1).

subroutine. (1) A sequence of instructions whose execution is invoked by a call. (2) A sequenced set of instructions or statements that can be used in one or more computer programs and at one or more points in a computer program. (3) A group of instructions that can be part of another routine or can be called by another program or routine.

synchronization. The action of forcing certain points in the execution sequences of two or more asynchronous procedures to coincide in time.

system administrator. (1) The person at a computer installation who designs, controls, and manages the use of the computer system. (2) The person who is responsible for setting up, modifying, and maintaining the Parallel Environment.

T

target application. See *DPCL target application*.

task. A unit of computation analogous to a process. 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 single, separately dispatchable, unit of execution. There can be one or more threads in a process, and each thread is executed by the operating system concurrently.

TPD. Third party debugger.

tracing. In PE, the collection of information about the execution of the program. This information is accumulated into a trace file that can later be examined.

tracepoint. Tracepoints are places in the program that, when reached during execution, cause the debugger to print information about the state of the program.

trace record. In PE, a collection of information about a specific event that occurred during the execution of your program. For example, a trace record is created for each send and receive operation that occurs in your program (this is optional and might not be appropriate). These records are then accumulated into a trace file that can later be examined.

U

unrolling loops. See *loop unrolling.*

user. (1) A person who requires the services of a computing system. (2) Any person or any thing that can issue or receive commands and message to or from the information processing system.

User Space. A version of the message passing library that is optimized for direct access to the high performance switch. User Space maximizes performance by passing up all kernel involvement 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. (1) In programming languages, a named object that may take different values, one at a time. The values of a variable are usually restricted to one data type. (2) A quantity that can assume any of a given set of values. (3) A name used to represent a data item whose value can be changed while the program is running. (4) A name used to represent data whose value can be changed, while the program is running, by referring to the name of the variable.

X

X Window System. The UNIX industry's graphics windowing standard that provides simultaneous views of several executing programs or processes on high resolution graphics displays.

Index

Reader’s Comments– We’d like to hear from you

IBM Parallel Environment for AIX 5L
Introduction
Version 4 Release 3.0

Publication No. SA22-7947-05

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
SA22-7947-05

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 55JA, Mail Station P384
2455 South Road
Poughkeepsie NY
12601-5400

Fold and Tape | **Please do not staple** | Fold and Tape

SA22-7947-05

Cut or Fold
Along Line

Program Number: 5765-F83

SA22-7947-05