IBM Parallel Environment for AIX: Hitchhiker's Guide Version 2, Release 3

Document Number GC23-3895-02

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vi.

Second Edition, August 1997

This edition applies to Version 2, Release 3, Modification 0 of the IBM IBM Parallel Environment for AIX (5765-543), and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

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
522 South Road
Poughkeepsie, N.Y. 12601-5400
United States of America

FAX (United States and Canada): 1+914+432-9405
FAX (Other Countries):
Your International Access Code +1+914+432-9405

IBMLink (United States customers only): KGNVMC(MHVRCFS)
IBM Mail Exchange: USIB6TC9 at IBMMAIL
Internet e-mail: mhvrcfs@vnet.ibm.com
World Wide Web: http://www.rs6000.ibm.com (select Parallel Computing)

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.

Copyright International Business Machines Corporation 1995, 1997. All rights reserved. Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp..

Permission to copy without fee all or part of *MPI: A Message Passing Interface Standard, Version 1.1*, Message Passing Interface Forum is granted, provided the University of Tennessee copyright notice and the title of the document appear, and notice is given that copying is by permission of the University of Tennessee. ©1993, 1995 University of Tennessee, Knoxville, Tennessee.

Contents

Notices . . . vi
Trademarks . . . vi

About This Book . . . ix
The Place for This Book in the Grand Scheme of Life, the Universe, and Everything... . . . ix
What's All This? . . . ix
What You Should Have While You Read this Book . . . xi
Document Conventions . . . xi
Encyclopedia Galactica . . . xii
IBM Parallel Environment for AIX Publications . . . xii
Related IBM Publications . . . xiii
Related Non-IBM Publications . . . xiii
Ordering Additional Copies of This Book . . . xiii
Getting the Examples in this Book . . . xiii
National Language Support . . . xiv
Accessing Online Information . . . xiv
Obtaining Documentation . . . xv

Chapter 1. Hitching a Lift on the Vogon Constructor Ship . . . 1
What's the IBM Parallel Environment for AIX? . . . 1
What's the Parallel Operating Environment? . . . 2
Before You Start . . . 2
Installation . . . 2
Access . . . 3
AIX-Based User Authorization . . . 3
DFS/DCE-Based User Authorization . . . 4
Host List File . . . 4
Running POE . . . 5
Some Examples of Running POE . . . 6
Running POE with Environment Variables . . . 8
Compiling (a Little Vogon Poetry) . . . 10
Who's In Control (SP Users Only)? . . . 12
How Are Your SP Nodes Allocated? . . . 13
Getting a Little More Information . . . 14
POE Options . . . 19

Chapter 2. The Answer is 42 . . . 21
Message Passing . . . 22
Data Decomposition . . . 22
Functional Decomposition . . . 29
Duplication Versus Redundancy . . . 32
Protocols Supported . . . 33

Chapter 3. Don't Panic . . . 35
Messages . . . 35
Message Catalog Errors . . . 35
Finding PE Messages . . . 36
Logging POE Errors to a File . . . 36
Message Format . . . 36

© Copyright IBM Corp. 1997

Diagnosing Problems Using the Install Verification Program 37
Can't Compile a Parallel Program 37
Can't Start a Parallel Job 37
Can't Execute a Parallel Program 39
The Program Runs But... 41
The Parallel Debugger is Your Friend 41
It Core Dumps 42
No Output at All 47
It Hangs 48
Using the VT Displays 52
Let's Attach the Debugger 55
Other Hangups 62
Bad Output 62
Debugging and Threads 63
Keeping an Eye on Progress 64

Chapter 4. So Long And Thanks For All The Fish 67
Tuning the Performance of a Parallel Application 67
How Much Communication is Enough? 68
Tuning the Performance of Threaded Programs 72
Why is this so slow? 73
Profile it 74
Parallelize it 78
Wrong answer! 80
Here's the Fix! 85
It's Still Not Fast Enough! 87
Tuning Summary 89

Chapter 5. Babel fish 91
Point-to-Point Communication 91
SEND (Non-Blocking) 91
RECEIVE (Non-Blocking) 91
SEND (Blocking) 91
RECEIVE (Blocking) 92
SEND/RECEIVE (Blocking) 92
STATUS 92
WAIT 92
TASK_SET 92
TASK_QUERY 93
ENVIRON 93
STOPALL 94
PACK 94
UNPACK 94
VSEND (Blocking) 94
VRECV (Blocking) 94
PROBE 95
Collective Communications 95
BROADCAST 95
COMBINE 95
CONCAT 95
GATHER 95
INDEX 96
PREFIX 96
REDUCE 96

SCATTER . . . 96
SHIFT . . . 96
SYNC . . . 97
GETLABEL . . . 97
GETMEMBERS . . . 97
GETRANK . . . 97
GETSIZE . . . 97
GETTASKID . . . 98
GROUP . . . 98
PARTITION . . . 98
Reduction Functions . . . 98
User-Defined Reduction Functions . . . 99
Global Variables and Constants . . . 99
Last Error Code . . . 99
Wildcards . . . 99
General Notes . . . 100
Task Indentifiers . . . 100
Message Length . . . 100
Creating MPI Objects . . . 100
Using Wildcard Receives . . . 100
Reduction Functions . . . 101
Error Handling . . . 101
Mixing MPL and MPI Functions in the Same Application . . . 101
Before and After Using MPI Functions . . . 102
Using Message Passing Handlers . . . 102

Appendix A. A Sample Program to Illustrate Messages . . . 103
Figuring Out What All of This Means . . . 105

Appendix B. MPI Safety . . . 107
Safe MPI Coding Practices . . . 107
What's a Safe Program? . . . 107
Some General Hints and Tips . . . 108
Order . . . 109
Progress . . . 109
Fairness . . . 110
Resource Limitations . . . 110

Appendix C. Installation Verification Program Summary . . . 113
Steps Performed by the POE Installation Verification Program . . . 113

Appendix D. Parallel Environment Internals . . . 115
What Happens When I Compile My Applications? . . . 115
How Do My Applications Start? . . . 115
How Does POE Talk to the Nodes? . . . 116
How are Signals Handled? . . . 116
What Happens When My Application Ends? . . . 116

Glossary of Terms and Abbreviations . . . 117

Index . . . 127

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries, or both:

AIX	IBM
RS/6000	RS/6000 Scalable POWERparallel Systems
SP	POWERparallel

X-Windows System is a trademark of the Massachusetts Institute of Technology

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

PostScript is a trademark of Adobe Systems, Incorporated

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Some of the references and quotes herein are from the *The Hitchhiker's Guide to the Galaxy*, by Douglas Adams, used by permission. ©1986 by Douglas Adams.

Some of the references and quotes herein are from *MPI: A Message-Passing Interface Standard, Version 1.1*, Message Passing Interface Forum, June 6, 1995. Permission to copy *MPI: A Message-Passing Interface Standard, Version 1.1*, Message Passing Interface Forum, is granted, provided the University of Tennessee copyright notice and the title of the document appear, and notice is given that copying is by permission of the University of Tennessee. ©1993, 1995 University of Tennessee, Knoxville, Tennessee.

About This Book

This book provides suggestions and guidance for using the IBM Parallel Environment for AIX program product, version 2.3.0, to develop and run FORTRAN and C parallel applications. To make this book a little easier to read, the name *IBM Parallel Environment for AIX* 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, Visualization Tool, and Parallel Debugger.

This book has been written with references and allusions to *The Hitchhiker's Guide to the Galaxy*, by Douglas Adams. If you are not familiar with the book, the basic premise is an all-encompassing guidebook to the entire galaxy. While we don't have such grandiose aspirations for this book, we do expect it to assist you in finding your way around the IBM Parallel Environment for AIX.

The Place for This Book in the Grand Scheme of Life, the Universe, and Everything...

If you are new to either message passing parallel programming or to the IBM Parallel Environment for AIX, you should find one or more sections of this book useful. To make the best use of this book, you should be familiar with:

- The AIX operating system
- One or more of the supported programming languages (FORTRAN or C)
- Basic parallel programming concepts.

This book is not intended to provide comprehensive coverage of the topics above, nor is it intended to tell you everything there is to know about the PE program product. If you need more information on any of these topics, the publications listed in “Encyclopedia Galactica” on page xii may be of some help.

On the other hand, if you're beyond the novice level in one or more of the topics covered in this book, you may want to skip over the associated sections. In this case, you can refer to “Encyclopedia Galactica” on page xii for publications that might cover these topics in more depth.

What's All This?

In *The Hitchhiker's Guide to the Galaxy*, a novel by Douglas Adams, there are many situations and states that are similar to those experienced by a person that is becoming familiar with PE. That's why we fashioned this book after Mr. Adams's....the *parallels* were obvious. We decided to include these references to make reading this book a little more pleasant. We hope you enjoy it. If not, please let us know by submitting the Readers' Comment form at the back of this manual, or by sending us comments electronically (see the Edition Notice on page ii for directions on how to do this). Since *The Hitchhiker's Guide to the Galaxy* material in this book is not technical in nature, we will not open customer-reported APARs (Authorized Program Analysis Reports) related to it.

© Copyright IBM Corp. 1997

The chapter titles in this book are taken directly from references within *The Hitchhiker's Guide to the Galaxy*. For those unfamiliar with Mr. Adams' work, or if our references are less than obvious, the objective of each chapter is described below (so you don't think we've all gone completely mad).

- **Chapter 1, "Hitching a Lift on the Vogon Constructor Ship" on page 1** familiarizes you with the Parallel Operating Environment (POE). *The Hitchhiker's Guide to the Galaxy* begins with Arthur Dent, earthman and main character, being suddenly swept aboard an alien space ship; the Vogon Constructor Ship. Once on board the ship, Arthur is completely bewildered, the way you must feel right now if you're completely new to the IBM Parallel Environment for AIX and don't have any idea where to start.
- **Chapter 2, "The Answer is 42" on page 21** covers parallelization techniques and discusses their advantages and disadvantages. *The Hitchhiker's Guide to the Galaxy* tells us that the galaxy's biggest supercomputer was asked to come up with an answer to the ultimate question of *Life, the Universe, and Everything*. The answer was 42. The problem is that once the characters in the book have the answer, they realize they don't know what the question is. We've used this title for the chapter that discusses how you take a working serial program (you know the answer is the serial algorithm) and create a parallel program that gives the same answer (you need to determine what the parallel constructs are to implement the algorithm).
- **Chapter 3, "Don't Panic" on page 35** outlines the possible causes for a parallel application to fail to execute correctly, and how the tools available with the IBM Parallel Environment for AIX can be used to identify and correct problems. What do you do when your parallel program doesn't work right...and how many different ways are there for it not to work right? As *The Hitchhiker's Guide to the Galaxy* advises us, **Don't Panic**.
- **Chapter 4, "So Long And Thanks For All The Fish" on page 67** discusses some of the ways you can optimize the performance of your parallel program. In *The Hitchhiker's Guide to the Galaxy*, we learn that dolphins are the most intelligent life form on Earth. *So long and thanks for all the fish* is their departing message to mankind as they leave Earth. We're not leaving earth, but we'll leave you with some parting hints on tuning the performance of your program.
- **Chapter 5, "Babel fish" on page 91** helps you understand how to translate your MPL parallel program into a program that conforms to the MPI standard. In *The Hitchhiker's Guide to the Galaxy*, the *Babel Fish* is a tiny fish that, when inserted into your ear, can make any language understandable to you. It would be nice if we could give you a Babel Fish to migrate your MPL applications to MPI, but that technology is still a few billion years away.
- **Appendix A, "A Sample Program to Illustrate Messages" on page 103** 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, "MPI Safety" on page 107** provides you with some general guidelines for creating *safe* parallel MPI programs.
- **Appendix C, "Installation Verification Program Summary" on page 113** describes how the *POE Installation Verification Program*, helps you determine if your system was properly installed.

- **Appendix D, "Parallel Environment Internals" on page 115** provides some additional information about how the IBM Parallel Environment for AIX (PE) works, with respect to the user's application.

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

What You Should Have While You Read this Book

Although you can get some basic information by reading this book by itself, you'll get a lot more out of it if you use it during an actual parallel session. To do this, you'll need one of the following:

- A workstation with PE, version 2.3.0, installed.
- A cluster of workstations with PE, version 2.3.0, installed.
- An IBM RS/6000 SP (SP) machine with PE, version 2.3.0, installed.

The PE code samples in this book are available via anonymous FTP. These will also be useful as you go through this book. See "Getting the Examples in this Book" on page xiii for information on accessing them via the Internet.

It's probably a good idea to get the PE and other IBM manuals listed in "Encyclopedia Galactica" on page xii before you start. Depending on your level of expertise, you may want to look at one or more of the other books listed in "Encyclopedia Galactica" on page xii as well.

Document Conventions

This manual uses visual cues to help you locate and identify information quickly and easily. The highlighting conventions used in this book are outlined in the following table.

Typographic	Usage
Bold	• **Bold** words or characters represent system elements that you must use literally, such as commands, flags, and path names. • **Bold** words also indicate the first use of a term included in the glossary.
Italic	• *Italic* words or characters represent variable values that you must supply. • *Italics* are also used for book titles and for general emphasis in text.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices. (In other words, it means "or.")
< >	Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, <**Enter**> refers to the key on your terminal or workstation that is labeled with the word Enter.
...	An ellipsis indicates that you can repeat the preceding item one or more times.
<**Ctrl-***x*>	The notation <**Ctrl-***x*> indicates a control character sequence. For example, <**Ctrl-c**> means that you hold down the control key while pressing <**c**>.

How Tasks Are Documented

This manual presents instructions for performing tasks. User actions appear in uppercase boldface type. For example, if the action is to enter the command **tool**, this manual presents the instruction as:

ENTER tool

The system response to an action follows a triangle (▲). So if the Tool Main Window opens in response to your entering the **tool** command, this manual presents it as:

▲ The Tool Main Window appears.

Selections from a menu bar are indicated with an arrow (→). For example, this manual would present an instruction to select the option **Yes** from the pull-down menu **Quit** as:

SELECT Quit → Yes

Angle brackets are sometimes used to enclose the name of keys on the keyboard. In particular, this is used to indicate a control-character sequence. For example, an instruction to hold the control key down while pressing “c” on the keyboard would be:

PRESS <**Ctrl-c**>

The examples in this book assume that you are using the Korn shell. An instruction to set an environment variable, for example, will use the **export** command.

Encyclopedia Galactica

IBM Parallel Environment for AIX Publications

- *IBM Parallel Environment for AIX: General Information* (GC23-3906)
- *IBM Parallel Environment for AIX: Installation Guide*, (GC28-1981)
- *IBM Parallel Environment for AIX: Operation and Use, Vol. 1*, (SC28-1979)
- *IBM Parallel Environment for AIX: Operation and Use, Vol. 2*, (SC28-1980)
- *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*, (GC23-3894)
- *IBM Parallel Environment for AIX: Messages*, (GC28-1982)
- *IBM Parallel Environment for AIX: Licensed Program Specification*, (GC23-3896)

As an alternative to ordering the individual books, you can use SBOF-8588 to order the entire IBM Parallel Environment for AIX library.

Related IBM Publications

- *IBM AIX Version 3.2 and 4.1 Performance Monitoring and Tuning Guide*, (SC23-2365)
- *AIX for RISC System/6000: Optimization and Tuning Guide for FORTRAN, C and C++*, (SC09-1705)
- *IBM XL FORTRAN Compiler for AIX: Users Guide*, (SC09-1610)
- *IBM XL FORTRAN Compiler for AIX: Language Reference*, (SC09-1611)
- *C Set ++ for AIX/6000: Language Reference*, (SC09-1606)
- *C Set ++ for AIX/6000: Standard Class Library Reference*, (SC09-1604)
- *C Set ++ for AIX/6000: User's Guide*, (SC09-1605)
- *C Set ++ for AIX/6000: C Language Reference*, (SC09-1730)

Related Non-IBM Publications

- Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 1.1*, University of Tennessee, Knoxville, Tennessee, June 6, 1995.
- Almasi, G., Gottlieb, A. *Highly Parallel Computing*. Benjamin-Cummings Publishing Company, Inc., 1989.
- Foster, I., *Designing and Building Parallel Programs*. Adison Wesley, 1995.
- Gropp, W., Lusk, E., Skjellum, A. *Using MPI*. The MIT Press, 1994.
- Bergmark, D., Pottle, M. *Optimization and Parallelization of a Commodity Trade Model for the SP1*. Cornell Theory Center, Cornell University, June, 1994.
- Phister, Gregory, F., *In Search of Clusters* Prentice Hall, 1995.
- Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J. *MPI: The Complete Reference* The MIT Press, 1996.
- Spiegel, Murray, R., *Vector Analysis* McGraw-Hill, 1959.

Ordering Additional Copies of This Book

To order additional copies of this book, use order number GC23-3895.

To order the full set of PE books, use order number SBOF-8588.

Getting the Examples in this Book

All of the serial and parallel programs used as examples in this book are available via anonymous FTP. Instructions for accessing these programs from the FTP server are in the README files that are included on the PE product tape. The PE product README files are:

- /usr/lpp/ppe.poe/README/poe.README
- /usr/lpp/ppe.vt/README/vt.README
- /usr/lpp/ppe.pedb/README/pedb.README
- /usr/lpp/ppe.xprofiler/README/xprofiler.README
- /usr/lpp/ppe.pedocs/README/pedocs.README

The README files also contain information on accessing the IBM RS/6000 SP World Wide Web site and home page, which contain all the books in the IBM Parallel Environment for AIX library in browsable and downloadable format. It also contains the serial and parallel programs mentioned above. You need to have access to the World Wide web in order to use the IBM RS/6000 SP website.

The examples in this book were placed into subdirectories on the FTP server according to the chapter in which they appear. The HitchhikersGuidePE.README file, on the FTP server, lists these examples and the corresponding subdirectories in which they are found. This README file is also part of the **aixpe_examples.tar** file on the server.

National Language Support

For National Language Support (NLS), all PE components and tools display messages located in externalized message catalogs. English versions of the message catalogs are shipped with the PE program product, but your site may be using its own translated message catalogs. The AIX environment variable **NLSPATH** is used by the various PE components 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 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 additional information on NLS and message catalogs, see *IBM Parallel Environment for AIX: Messages* and *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

Accessing Online Information

The PE program product provides manual pages as well as online versions of the publications that can be viewed using an HTML browser (such as Netscape, WebExplorer, or Mosaic).

To access a list of manual pages for PE:

ENTER **man pe**

Note: In order to use the PE MPI subroutine manual pages or access the PE online (HTML) publications, the **ppe.pedocs** fileset must first be installed. To view the PE online publications, you need the following:

- Access to a common HTML document browser (such as Netscape, WebExplorer, or Mosaic)
- The location of the HTML index file, provided with the **ppe.pedocs** fileset. Contact your system administrator or installer for this location.

Obtaining Documentation

This book is available in PostScript format. You can obtain the PostScript files for the product documentation by:

- Downloading them from the **lscftp.pok.ibm.com** anonymous FTP server. The files for **PE_prod_name** can be found in the directory called /**pub**/**pps**/**PE**.
- Printing the PostScript documentation files that are shipped on the product media.

You can also view or download this book from the IBM RS/6000 web site, whose current URL is listed in the PE filesets' README files.

Chapter 1. Hitching a Lift on the Vogon Constructor Ship

The Hitchhiker's Guide to the Galaxy begins with Arthur Dent, earthman and main character, being suddenly swept aboard an alien spaceship from his garden. Once on the ship, Arthur is totally bewildered by his new surroundings. Fortunately, Ford Prefect, Arthur's earth companion (who, as Arthur recently discovered, is actually an alien from a planet somewhere in the Betelguese system), is there to explain what's going on.

Just as Arthur had to get used to his new environment when he first left earth, this chapter will help you get used to the new environment you're in; the IBM Parallel Environment for AIX (PE). It covers:

- The IBM Parallel Environment for AIX
- The Parallel Operating Environment
- Starting the POE
- Running simple commands
- Experimenting with parameters and environment variables
- Using a *host list* file versus the Resource Manager for requesting processor nodes
- Compiling and running a simple parallel application
- Some simple environment setup and debugging tips.

This book contains many examples and illustrates various commands and programs as well as the output you get as a result of running them. When looking at these examples, please keep in mind that the output you see on your system may not exactly match what's printed in the book, due to the differences between your system and ours. We've included them here just to give you a basic idea of what happens.

The sample programs, as they appear in this book, are also provided in source format on the anonymous FTP server (as described in “Getting the Examples in this Book” on page xiii). If you intend to write or use any of the sample programs, it would be best if you obtained the examples from the FTP site rather than copying them from the book. Because of formatting and other restrictions in publishing code examples, some of what you see here may not be syntactically correct. On the other hand, the source code at the FTP site will work (we paid big bucks to someone to make sure they did).

If you're unfamiliar with the terms in this chapter, the “Glossary of Terms and Abbreviations” on page 117 may be of some help.

What's the IBM Parallel Environment for AIX?

The IBM Parallel Environment for AIX (PE) software lets you develop, debug, analyze, tune, and execute parallel applications, written in FORTRAN, C, and C++, quickly and efficiently. PE conforms to existing standards like UNIX and MPI. PE runs on either an IBM RS/6000 SP (SP) machine, or an AIX workstation cluster.

PE consists of:

- The Parallel Operating Environment (POE), for submitting and managing jobs.

© Copyright IBM Corp. 1997

- Message passing libraries (MPL and MPI), for communication among the tasks that make up a parallel program.
- Parallel debuggers, for debugging parallel programs.
- The Visualization Tool (VT), for examining the communication and performance characteristics of a parallel program.
- Parallel utilities, for easing file manipulation.
- Xprofiler, for analyzing a parallel application's performance.

What's the Parallel Operating Environment?

The purpose of the Parallel Operating Environment (POE) is to allow you to develop and execute your parallel applications across multiple processors, called *nodes*. When using POE, there is a single node (a workstation) called the *home node* that manages interactions with users.

POE transparently manages the allocation of remote nodes, where your 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 (more on that in a moment), but once you have, POE does the rest.

When we say *processor node*, we're talking about a physical entity or location that's defined to the network. It can be a standalone machine, or a processor node within an IBM RS/6000 SP (SP) frame. From POE's point of view, a node is a node...it doesn't make much difference.

If you're using an SMP system, it's important to know that although an SMP node has more than one processing unit, it is still considered, and referred to as, a *processor node*.

Before You Start

Before getting underway, you should check to see that the items covered in this section have been addressed.

Installation

Whoever installed POE (this was probably your System Administrator but may have been you or someone else) should verify that it was installed successfully by running the *Installation Verification Program* (IVP). The IVP, which verifies installation for both threaded and non-threaded environments, is discussed briefly in Appendix C, “Installation Verification Program Summary” on page 113, and is covered in more detail in *IBM Parallel Environment for AIX: Installation Guide*.

The IVP tests to see if POE is able to:

- Establish a remote execution environment
- Compile and execute your program
- Initialize the IP message passing environment.

- Check that both the signal-handling and threaded versions of the MPI library are operable.

The instructions for verifying that the PE Visualization Tool (VT) was installed correctly are in *IBM Parallel Environment for AIX: Installation Guide*.

Access

Before you can run your job, you must first have access to the compute resources in your system. Here are some things to think about:

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

POE allows two types of user authorization:

1. AIX based user authorization, using entries in /**etc**/**hosts.equiv** or **.rhosts** files. This is the default POE user authorization method.
2. DFS/DCE-based user authorization, using DCE credentials. If you plan to run POE jobs in a DFS environment, you must use DFS/DCE-based user authorization.

The type of user authorization is controlled by the MP_AUTH environment variable. The valid values are **AIX** (the default) or **DFS**.

The system administrator can also define the value for MP_AUTH in the /**etc**/**poe.limits** file. If MP_AUTH is specified in /**etc**/**poe.limits**, POE overrides the value of the MP_AUTH environment variable if it's different.

AIX-Based User Authorization

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

- Authorize both the home node machine and your user name (or machine names) in the /**etc**/**hosts.equiv** file on each remote node.

  or
- Set up a **.rhosts** file in the home directory of your 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 for AIX: Installation Guide

/**etc**/**hosts.equiv** is checked first, and if the home node and user/machine name don't 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 *202r1n10*, try the following command:

```
$ rsh <202r1n10> hostname
```

The response to this should be the remote host name. If it isn't the remote host name, or the command cannot be run, you'll have to see your system administrator.

Issue this command for every remote host on which you plan to have POE execute your job.

Refer to *IBM Parallel Environment for AIX: Installation Guide* for more detailed information.

DFS/DCE-Based User Authorization

If you plan to run POE on a system with the Distributed File System (DFS), you need to perform some additional steps in order to enable POE to run with DFS.

DFS requires you to have a set of DCE credentials which manage the files to which you have access. Since POE needs access to your DCE credentials, you need to provide them to POE:

1. Do a **dce_login**. This enables your credentials through DCE (ensures that you are properly authenticated to DCE).
2. Propagate your credentials to the nodes on which you plan to run your POE jobs. Use the **poeauth** command to do this. **poeauth** copies the credentials from the home node, using a host list file or Resource Manager pool. The first node in the host list or pool (the home node) *must* be the node from which you did the **dce_login** (and is where the credentials exist).

   poeauth is actually a POE application program that is used to copy the DCE credentials. As a result, before you attempt to run **poeauth** on a DFS system, you need to make your current working directory a non-DFS directory (for example, /tmp). Also, if you use a host list file, it must be in a non-DFS directory. Otherwise, you may encounter errors running **poeauth** which are related to POE's access of DFS directories.

Keep in mind that each node in your system, on which parallel jobs may run, requires the DFS/DCE credentials. As a result, it's wise to use the **poeauth** command with a host list file or pool that contains every node on which might want to run your jobs later.

DCE credentials are maintained on a per user basis, so each user will need to invoke **poeauth** themselves in order to copy the credentials files. The credentials remain in effect on all nodes to which then were copied until they expire (at which time you will need to copy them using **poeauth** again).

For more information on running in a DFS environment and the **poeauth** command, refer to *IBM Parallel Environment for AIX: Operation and Use, Vol. 1*.

Host List File

One way to tell POE where to run your program is by using a *host list* file. The host list file is generally in your 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 (as we'll discuss later). This file contains one of two different kinds of information; node names or pool numbers. Note that your host list file cannot contain a mixture of node names and pool numbers, so you must specify one or the other.

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

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

Pools are groups of nodes that are known to the SP *Resource Manager*, and identified by a number. Pools are generally defined by the SP system administrator, and then assigned to particular groups of people. They are entered in the host list file with an *at*(@) sign, followed by the pool number (for instance, @1).

Here's an example of a host list file that specifies pool numbers. Four tasks will run; two on nodes in pool 10 and two on nodes in pool 11.

```
@10
@10
@11
@11
```

Running POE

Once you've checked all the items in “Before You Start” on page 2, you're ready to run the Parallel Operating Environment. At this point, you can view POE as a way to run commands and programs on multiple nodes from a single point. It is important to remember that these commands and programs are really running on the remote nodes, and if you ask POE to do something on a remote node, everything necessary to do that thing must be available on that remote node. More on this in a moment.

Note that there are two ways to influence the way your parallel program is executed; with environment variables or command-line option flags. You can set environment variables at the beginning of your session to influence each program that you execute. You could also get the same effect by specifying the related command-line flag when you invoke POE, but its influence only lasts for that particular program execution. For the most part, this book shows you how to use the command-line option flags to influence the way your program executes. “Running POE with Environment Variables” on page 8 gives you some high-level information, but you may also want to refer to *IBM Parallel Environment for AIX: Operation and Use, Vol. 1* to learn more about using environment variables.

One more thing. In the following sections, we show you how to run POE by requesting nodes via a host list file. Note, however, that you may also request nodes via the Resource Manager. The Resource Manager is covered in more detail in “Who's In Control (SP Users Only)?” on page 12, as well as *IBM Parallel Environment for AIX: Operation and Use, Vol. 1*.

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 your program and reproduces your 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 an AIX command. When you try these examples on your system, use a host list file that contains the node names (as opposed to a pool number). The reason for this will be discussed a little later. These examples also 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, it's okay to duplicate lines. This example assumes your file is called *host.list*, and is in the directory from which you're 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 (covered later....one thing at a time!).

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.pok.ibm.com
 202r1n11.hpssl.pok.ibm.com
 202r1n09.hpssl.pok.ibm.com
 202r1n12.hpssl.pok.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 your workstation. One thing that you don't see is which task is responsible for each line of output. In a simple example like this, it isn't that important but if you had many lines of output from each node, you'd 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.pok.ibm.com
  2:202r1n11.hpssl.pok.ibm.com
  0:202r1n09.hpssl.pok.ibm.com
  3:202r1n12.hpssl.pok.ibm.com
```

This time, notice how each line starts with a number and a colon? Notice also 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). Now we can 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 4.2   on pe03.pok.ibm.com                       *
0:*                                                                             *
0:*******************************************************************************
0:*                                                                             *
0:*     Message of the Day:  Never drink more than 3 Pan                        *
0:*     Galactic Gargle Blasters unless you are a 50 ton maga                   *
0:*     elephant with nemona.                                                   *
0:*                                                                             *
1:*******************************************************************************
1:*                                                                             *
1:*  Welcome to IBM AIX Version 4.2   on pe04.pok.ibm.com                       *
1:*                                                                             *
1:*******************************************************************************
1:*                                                                             *
1:*                                                                             *
1:*     Message of the Day:  Never drink more than 3 Pan                        *
1:*     Galactic Gargle Blasters unless you are a 50 ton maga                   *
1:*     elephant with nemona.                                                   *
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. But 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 your 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, we can ask POE to 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 4.2   on pe03.pok.ibm.com                       *
0:*                                                                             *
0:*******************************************************************************
0:*                                                                             *
0:*                                                                             *
0:*                                                                             *
0:*     Message of the Day:  Never drink more than 3 Pan                        *
0:*     Galactic Gargle Blasters unless you are a 50 ton maga                   *
0:*     elephant with nemona.                                                   *
0:*                                                                             *
0:*                                                                             *
0:*                                                                             *
0:*******************************************************************************
```

```
1:*******************************************************************************
1:*                                                                             *
1:*  Welcome to IBM AIX Version 4.2   on pe04.pok.ibm.com                       *
1:*                                                                             *
1:*******************************************************************************
1:*                                                                             *
1:*                                                                             *
1:*                                                                             *
1:*     Message of the Day:  Never drink more than 3 Pan                        *
1:*     Galactic Gargle Blasters unless you are a 50 ton maga                   *
1:*     elephant with nemona.                                                   *
1:*                                                                             *
1:*                                                                             *
1:*                                                                             *
1:*******************************************************************************
```

This time, POE holds onto 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 your program, but it's easier to understand. Note that the **-stdoutmode** option consumes a significant amount of system resources, which may affect performance.

Running POE with Environment Variables

By the way, if you're getting tired of typing the same command line options over and over again, you can set them as environment variables so you don't 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's **MP_LABELIO**. If we set these two variables like this:

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

we can then run our /**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 4.2   on pe03.pok.ibm.com                       *
0:*                                                                             *
0:*******************************************************************************
0:*                                                                             *
0:*                                                                             *
0:*                                                                             *
0:*     Message of the Day:  Never drink more than 3 Pan                        *
0:*     Galactic Gargle Blasters unless you are a 50 ton maga                   *
0:*     elephant with nemona.                                                   *
0:*                                                                             *
0:*                                                                             *
0:*                                                                             *
0:*******************************************************************************
1:*******************************************************************************
1:*                                                                             *
```

```
1:*  Welcome to IBM AIX Version 4.2   on pe03.pok.ibm.com                    *
1:*                                                                          *
1:****************************************************************************
1:*                                                                          *
1:*                                                                          *
1:*                                                                          *
1:*     Message of the Day:  Never drink more than 3 Pan                     *
1:*     Galactic Gargle Blasters unless you are a 50 ton maga                *
1:*     elephant with nemona.                                                *
1:*                                                                          *
1:*                                                                          *
1:*                                                                          *
1:****************************************************************************
```

In the example above, notice how the program ran with two processes, and the output was labeled?

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

```
$ poe hostname

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

Notice how the program still ran with two processes and you got labeled output?

Now let's try overriding the environment variables we just set. To do this, we'll use command line options when we run POE. Try running the following command:

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

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

This time, notice how the program ran with four processes and the output wasn't 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.pok.ibm.com
  1:202r1n10.hpssl.pok.ibm.com
```

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

Compiling (a Little Vogon Poetry)

All this is fine, but you probably have your own programs that you want to (eventually) run in parallel. We're going to talk in a little more detail in Chapter 2, “The Answer is 42” on page 21 about creating parallel programs, but right now we'll cover compiling a program for POE. Almost any FORTRAN, C, or C++ program can be compiled for execution under POE.

According to *The Hitchhiker's Guide to the Galaxy*, Vogon poetry is the third worst in the Universe. In fact, it's so bad that the Vogons subjected Arthur Dent and Ford Prefect to a poetry reading as a form of torture. Some people may think that compiling a parallel program is just as painful as Vogon poetry, but as you'll see, it's really quite simple.

Before compiling, you should verify that:

- POE is installed on your system
- You are authorized to use POE
- A FORTRAN or C Compiler is installed on your system.

See *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference* for information on compilation restrictions for POE.

To show you how compiling works, we've selected the *Hello World* program. Here it is in C:

```
/******************************************************************************
*
* Hello World C Example
*
* To compile:
* mpcc -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");
    return(0);
  }
```

And here it is in FORTRAN:

```
c*********************************************************************
c*
c* Hello World FORTRAN Example
c*
c* To compile:
c* mpxlf -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 -o hello_world_c hello_world.c

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

mpcc, **mpCC**, and **mpxlf** are POE scripts that link the parallel libraries that allow your programs to run in parallel. **mpcc**, **mpCC** and **mpxlf** are for compiling non-threaded programs. Just as there is a version of the **cc** command called **cc_r**, that's used for threaded programs, there is also a script called **mpcc_r** (and also **mpxlf_r** and **mpCC_r**) for compiling threaded message passing programs. **mpcc_r** generates thread-aware code by linking in the threaded version of MPI, including the threaded VT and POE utility libraries. These threaded libraries are located in the same subdirectory as the non-threaded libraries.

All the parallel 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 for AIX: Operation and Use, Vol. 1*.

Running the **mpcc**, **mpCC**, **mpxlf**, **mpcc_r**, **mpCC_r**, or **mpxlf_r** script, as we've shown you, creates an executable version of your source program that takes advantage of POE. However, before POE can run your program, you need to make sure it's accessible on each remote node. You can do this by either copying it there, or by mounting the file system that your program is in to each remote node.

Here's 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's the output of the FORTRAN program:

```
$ poe hello_world_f -procs 4

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

Figure 1. Output from mpcc/mpxlf

Who's In Control (SP Users Only)?

So far, we've explicitly specified to POE the set of nodes on which to run our parallel application. We did this by creating a list of hosts in a file called *host.list*, in the directory from which we 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 we told POE to use (with the **-procs** option).

Another way to tell POE which hosts to use is with the Resource Manager of the IBM AIX Parallel System Support Programs. Note that if you're running the PE on a RS/6000 cluster of nodes, discussion of the Resource Manager doesn't apply to you, and you may want to skip ahead to “Getting a Little More Information” on page 14.

The Resource Manager is used to allocate nodes, one job at a time. This is necessary if your parallel application is communicating directly over the SP 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 for AIX: Operation and Use, Vol. 1* for more information. It's also a convenient mechanism for dedicating the parallel nodes to a single job, so there's no conflict or contention for resources. The Resource Manager allocates nodes from either the host list file, or from a predefined *pool*, which the System Administrator usually sets up.

Note that this section discusses only the basics of node allocation; it doesn't address performance considerations. See *IBM Parallel Environment for AIX: Operation and Use, Vol. 1* for information on maximizing your program's performance.

How Are Your SP Nodes Allocated?

So how do you know who's allocating the nodes and where they're being allocated from? First of all, you must always have a *host list* file (or use the **MP_RMPOOL** environment variable or **-rmpool** command line option). As we've already mentioned, 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
```

would use 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.

If the **myHosts** file contains *pool* IDs (for example, @1), the Resource Manager will allocate nodes from the set that the System Administrator defined as *Pool 1*. Note, however, that there's another way to designate the pool on which you want your program to run. If **myHosts** didn't contain any pool IDs, you could use the:

- **MP_RMPOOL** environment variable to specify a pool number. This setting would last for the duration of your session.
- **-rmpool** command line option to specify a pool number when you invoke your program. Note that this option would override the **MP_RMPOOL** environment variable.

Note: If a host list file is used, it will override anything you specify with the **MP_RMPOOL** environment variable or the **-rmpool** command line option. You must set **MP_HOSTFILE** or **-hostfile** to NULL in order for **MP_RMPOOL** or **-rmpool** to work

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

If the **myHosts** file contains actual host names, but you want to use the High Performance Switch directly for communication, the Resource Manager will only allocate the nodes that are listed in **myHosts**. The Resource Manager is used because only one parallel job at a time can use the switch directly, and the Resource Manager will make sure that a node is allocated to only one job at a time. If the host list file contains actual host names, but you don't want to use the High Performance Switch directly for communication, POE allocates the nodes from those listed in the host list file.

As we said before, you can't have both host names and pool IDs in the same host list file.

Your program executes exactly the same way, regardless of whether POE or the Resource Manager allocated the nodes. In the following example, the host list file contains a pool ID which causes the Resource Manager to allocate nodes. However, the output is identical to Figure 1 on page 12, 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 your nodes are allocated, how do you skeptics know whether the Resource Manager was really used? Well, POE knows a lot that it ordinarily doesn't tell you. If you coax it with the **-infolevel** option, POE will tell you more than you ever wanted to know. Read on...

Getting a Little More Information

You can control the level of messages you get from POE as your 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 your program executes. For example, with **-infolevel** set to 2, you see a couple of different things. First, you'll see a message that says POE has connected to the Resource Manager. Following that, you'll see messages that indicate which nodes the Resource Manager passed back to POE for use.

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

Here's the *Hello World* program again:

```
$ poe hello_world_c -procs 2 -hostfile pool.list -labelio yes -infolevel 2
```

You should see output similar to the following:

```
INFO: 0031-690  Connected to Resource Manager
INFO: 0031-118  Pool 1 requested for task 0
INFO: 0031-118  Pool 1 requested for task 1
INFO: 0031-119  Host k10n01.ppd.pok.ibm.com allocated for task 0
INFO: 0031-119  Host k10n02.ppd.pok.ibm.com allocated for task 1
  0:INFO: 0031-724  Executing program: <hello_world_c>
  1:INFO: 0031-724  Executing program: <hello_world_c>
  1:Hello, World!
  0:Hello, World!
  0:INFO: 0033-3075 VT Node Tracing completed.  Node merge beginning
  1:INFO: 0033-3075 VT Node Tracing completed.  Node merge beginning
  0:INFO: 0031-306  pm_atexit: pm_exit_value is 0.
  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 STDOUT closed by task 1
INFO: 0031-656  I/O file STDERR closed by task 0
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're 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 we didn't set **-infolevel**, we would see only the output of the executable (Hello World!, in the example above), interspersed with POE output from remote nodes.

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

Look at the output, below. In this case, POE tells us that it's 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 -procs 2 -hostfile pool.list -labelio yes -infolevel 3
```

You should see output similar to the following:

```
INFO: DEBUG_LEVEL changed from 0 to 1
D1<L1>: Open of file pool.list successful
D1<L1>: mp_euilib = ip
D1<L1>: task 0 5 1
D1<L1>: extended 1  5 1
D1<L1>: node allocation strategy = 2
INFO: 0031-690  Connected to Resource Manager
INFO: 0031-118  Pool 1 requested for task 0
INFO: 0031-118  Pool 1 requested for task 1
D1<L1>: Elapsed time for call to jm_allocate: 1 seconds
INFO: 0031-119  Host k10n01.ppd.pok.ibm.com allocated for task 0
INFO: 0031-119  Host k10n02.ppd.pok.ibm.com allocated for task 1
D1<L1>: Requesting service pmv2
D1<L1>: Jobid = 803751678
  0:INFO: 0031-724  Executing program: <hello_world_c>
  1:INFO: 0031-724  Executing program: <hello_world_c>
  1:INFO: DEBUG_LEVEL changed from 0 to 1
  1:D1<L1>: mp_euilib is <ip>
  1:D1<L1>: mp_css_interrupt is <0>
  1:D1<L1>: cssAdapterType is <1>
D1<L1>: init_data for task 1: <129.40.161.66:1559>
  0:INFO: DEBUG_LEVEL changed from 0 to 1
  0:D1<L1>: mp_euilib is <ip>
  0:D1<L1>: mp_css_interrupt is <0>
  0:D1<L1>: cssAdapterType is <1>
D1<L1>: init_data for task 0: <129.40.161.65:1649>
  0:D1<L1>: About to call mpci_connect
  1:D1<L1>: About to call mpci_connect
  0:D1<L1>: Elapsed time for mpci_connect: 0 seconds
  1:D1<L1>: Elapsed time for mpci_connect: 0 seconds
  0:D1<L1>: About to call _ccl_init
  1:D1<L1>: About to call _ccl_init
  0:D1<L1>: Elapsed time for _ccl_init: 1 seconds
  0:Hello, World!
  1:D1<L1>: Elapsed time for _ccl_init: 0 seconds
  0:INFO: 0033-3075 VT Node Tracing completed.  Node merge beginning
  1:Hello, World!
  0:INFO: 0031-306  pm_atexit: pm_exit_value is 0.
  1:INFO: 0033-3075 VT Node Tracing completed.  Node merge beginning
  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 STDOUT closed by task 1
D1<L1>: Accounting data from task 0 for source 0:
D1<L1>: Accounting data from task 1 for source 1:
INFO: 0031-656  I/O file STDERR closed by task 0
```

```
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
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's happening on the home node, but if you want to see what's 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.***pid.taskid*,

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

If **-infolevel** is set high enough, the process number will be displayed in the output. If you don't know what the process number is, it's probably the most recent log file. If you're sharing the node with other POE users, the process number will be *one* of the most recent log files (but you own the file, so you should be able to tell).

Here's a sample log file:

```
AIX Parallel Environment pmd2 version @(#) 95/06/22 10:53:26
The ID of this process is 14734
The hostname of this node is k6n05.ppd.pok.ibm.com
The taskid of this task is 0
HOMENAME: k6n05.ppd.pok.ibm.com
USERID: 1063
USERNAME: vt
GROUPID: 1
GROUPNAME: staff
PWD: /u/vt/hughes
PRIORITY: 0
NPROCS: 2
PMDLOG: 1
NEWJOB: 0
PDBX: 0
AFSTOKEN: 5765-144 AIX Parallel Environment
LIBPATH: /usr/lpp/ppe.poe/lib:/usr/lpp/ppe.poe/lib/ip:/usr/lib
ENVC recv'd
envc: 23
envc is 23
env[0] = _=/bin/poe
env[1] = LANG=En_US
env[2] = LOGIN=vt
env[3] = NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
env[4] = PATH=/bin:/usr/bin:/etc:/usr/ucb:/usr/sbin:/usr/bin/X11:.:
env[5] = LC__FASTMSG=true
env[6] = LOGNAME=vt
env[7] = MAIL=/usr/spool/mail/vt
env[8] = LOCPATH=/usr/lib/nls/loc
env[9] = USER=vt
env[10] = AUTHSTATE=compat
env[11] = SHELL=/bin/ksh
env[12] = ODMDIR=/etc/objrepos
env[13] = HOME=/u/vt
env[14] = TERM=aixterm
env[15] = MAILMSG=[YOU HAVE NEW MAIL]
env[16] = PWD=/u/vt/hughes
```

```
env[17] = TZ=EST5EDT
env[18] = A__z=! LOGNAME
env[19] = MP_PROCS=2
env[20] = MP_HOSTFILE=host.list.k6
env[21] = MP_INFOLEVEL=2
env[22] = MP_PMDLOG=YES
Initial data msg received and parsed
Info level = 2
User validation complete
About to do user root chk
User root check complete
SSM_PARA_NODE_DATA msg recv'd
  0:129.40.84.69:k6n05.ppd.pok.ibm.com:  -1
  1:129.40.84.70:k6n06.ppd.pok.ibm.com:  -1
node map parsed
newjob is 0.
msg read, type is 13
string = <JOBID 804194891
hello_world_c >
SSM_CMD_STR recv'd
JOBID id 804194891
command string is <hello_world_c >
pm_putargs: argc = 1, k = 1
SSM_CMD_STR parsed
child pipes created
child: pipes successfully duped
child: MP_CHILD = 0
partition id is <31>
child: after initgroups (*group_struct).gr_gid = 100
child: after initgroups (*group_struct).gr_name = 1
fork completed
parent: my child's pid is 15248
attach data sent
pmd child: core limit is 1048576, hard limit is 2147483647
pmd child: rss limit is 33554432, hard limit is 2147483647
pmd child: stack limit is 33554432, hard limit is 2147483647
pmd child: data segment limit is 134217728, hard limit is 2147483647
pmd child: cpu time limit is 2147483647, hard limit is 2147483647
pmd child: file size limit is 1073741312, hard limit is 1073741312
child: (*group_struct).gr_gid = 1
child: (*group_struct).gr_name = staff
child: userid, groupid and cwd set!
child: current directory is /u/vt/hughes
child: about to start the user's program
child: argument list:
argv[0] = hello_world_c
argv[1] (in hex) = 0
child: environment:
env[0] = _=/bin/poe
env[1] = LANG=En_US
env[2] = LOGIN=vt
env[3] = NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
env[4] = PATH=/bin:/usr/bin:/etc:/usr/ucb:/usr/sbin:/usr/bin/X11:.:
env[5] = LC__FASTMSG=true
env[6] = LOGNAME=vt
env[7] = MAIL=/usr/spool/mail/vt
env[8] = LOCPATH=/usr/lib/nls/loc
env[9] = USER=vt
env[10] = AUTHSTATE=compat
env[11] = SHELL=/bin/ksh
env[12] = ODMDIR=/etc/objrepos
env[13] = HOME=/u/vt
env[14] = TERM=aixterm
env[15] = MAILMSG=[YOU HAVE NEW MAIL]
```

```
env[16] = PWD=/u/vt/hughes
env[17] = TZ=EST5EDT
env[18] = A__z=! LOGNAME
env[19] = MP_PROCS=2
env[20] = MP_HOSTFILE=host.list.k6
env[21] = MP_INFOLEVEL=2
env[22] = MP_PMDLOG=YES
child: LIBPATH = /usr/lpp/ppe.poe/lib:/usr/lpp/ppe.poe/lib/ip:/usr/lib
select: rc = 1
pulse is on, curr_time is 804180935, send_time is 0, select time is 600
pulse sent at 804180935
count = 51 on stderr
pmd parent: STDERR read OK:
STDERR: INFO: 0031-724  Executing program: <hello_world_c>
select: rc = 1
pulse is on, curr_time is 804180935, send_time is 804180935, select
time is 600
SSM type = 34
STDIN:
select: rc = 1
pulse is on, curr_time is 804180935, send_time is 804180935,
select time is 600
pmd parent: cntl pipe read OK:
pmd parent: type: 26, srce: 0, dest: -2, bytes: 6
parent: SSM_CHILD_PID: 15248
select: rc = 1
pulse is on, curr_time is 804180935, send_time is 804180935,
select time is 600
pmd parent: cntl pipe read OK:
pmd parent: type: 23, srce: 0, dest: -1, bytes: 18
select: rc = 1
pulse is on, curr_time is 804180935, send_time is 804180935,
select time is 600
SSM type = 29
STDIN: 129.40.84.69:1257
129.40.84.70:1213
select: rc = 1
pulse is on, curr_time is 804180935, send_time is 804180935,
select time is 600
pmd parent: cntl pipe read OK:
pmd parent: type: 44, srce: 0, dest: -1, bytes: 2
select: rc = 1
pulse is on, curr_time is 804180935, send_time is 804180935,
select time is 600
SSM type = 3
STDIN:
select: rc = 1
pulse is on, curr_time is 804180936, send_time is 804180935,
select time is 600
pmd parent: STDOUT read OK
STDOUT: Hello, World!
select: rc = 1
pulse is on, curr_time is 804180936, send_time is 804180935,
select time is 599
count = 65 on stderr
pmd parent: STDERR read OK:
STDERR: INFO: 0033-3075 VT Node Tracing completed.  Node merge beginning
select: rc = 1
pulse is on, curr_time is 804180936, send_time is 804180935,
select time is 599
count = 47 on stderr
pmd parent: STDERR read OK:
STDERR: INFO: 0031-306  pm_atexit: pm_exit_value is 0.
select: rc = 1
```

```
pulse is on, curr_time is 804180936, send_time is 804180935,
select time is 599
pmd parent: cntl pipe read OK:
pmd parent: type: 17, srce: 0, dest: -1, bytes: 2
select: rc = 1
pulse is on, curr_time is 804180936, send_time is 804180935,
select time is 599
SSM type = 5
STDIN: 5
select: rc = 1
pulse is on, curr_time is 804180936, send_time is 804180935,
select time is 599
in pmd signal handler
wait status is 00000000
exiting child pid = 15248
err_data is 0
select: rc = 2
pulse is on, curr_time is 804180936, send_time is 804180935,
select time is 599
count = 0 on stderr
child exited and all pipes closed
err_data is 0
pmd_exit reached!, exit code is 0
```

Appendix A, “A Sample Program to Illustrate Messages” on page 103 includes an example of setting **-infolevel** to 6, and explains the important lines of output.

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 for AIX: Operation and Use, Vol. 1*, but here are the ones you'll most likely need to be familiar with at this stage.

-procs

When you set **-procs**, you're telling POE how many tasks your 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 your program are labeled by task id. This becomes especially useful when you're running a parallel program and your output is *unordered*. With labeled output, you can easily determine which task returns which message.

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. Note that using the **-infolevel** option consumes a significant amount of system resources. 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 in /**tmp** on each of the remote nodes of your partition. These diagnostic logs are particularly useful for isolating the cause of abnormal termination. Note that the **-infolevel** option consumes a significant amount of system resources. Use it with care. You can also set the **MP_PMDLOG** environment variable to do this (**-pmdlog** can be used to temporarily override it).

-stdoutmode

The **-stdoutmode** option lets you specify how you want the output data from each task in your 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. We showed you how this works in some of the examples in this section. Note that using the **-infolevel** option consumes a significant amount of system resources. Use it with care. You can also set the **MP_STDOUTMODE** environment variable to do this (**-stdoutmode** can be used to temporarily override it).

Chapter 2. The Answer is 42

If you're familiar with message passing parallel programming, and you're completely familiar with message passing protocols, then you already know the answer is 42, and you can skip ahead to Chapter 3, “Don't Panic” on page 35 for a discussion on using the IBM Parallel Environment for AIX tools. If you're familiar with message passing parallel programming, (or if you can just say that five times really fast), but you would like to know more about the IBM Parallel Environment for AIX message passing protocols, look at the information in “Protocols Supported” on page 33 before skipping ahead to the next chapter.

For the rest of us, this section discusses some of the techniques for creating a parallel program, using message passing, as well as the various advantages and pitfalls associated with each.

This chapter is not intended to be an in-depth tutorial on writing parallel programs. Instead, it's more of an introduction to basic message passing parallel concepts; it provides just enough information to help you understand the material covered in this book. If you want more information about parallel programming concepts, you may find some of the books listed in “Related Non-IBM Publications” on page xiii helpful.

Good. Now that we've gotten rid of the know-it-alls that skipped to the next section, we can reveal the answer to the question of creating a successful parallel program. No, the answer isn't really 42. That was just a ploy to get everyone else out of our hair. The answer is to *start with a working sequential program*. Complex sequential programs are difficult enough to get working correctly, without also having to worry about the additional complexity introduced by parallelism and message passing. The bottom line is that it's 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'll develop an awareness of which sequential techniques translate better into parallel implementations, and you can then make a point of using these techniques in your sequential programs. In this chapter, you'll find information on 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 has to do with distributing the data that the program is processing among the parallel tasks. Each task does roughly 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 don't use data decomposition or functional decomposition exclusively, but rather 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

© Copyright IBM Corp. 1997

Message Passing

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. In the case of the SP, its 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 in order to interact. In this situation, tasks must exchange messages in order 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 is one way of boosting performance.

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

Data Decomposition

A good technique for parallelizing a sequential application 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 doesn't include the routines for computing the coefficient and determinant because they are not part of the parallelization at this point. Note also that this example is incomplete; you can get the entire program by following the directions in “Getting the Examples in this Book” on page xiii.

```
/*********************************************************************
 *
 * 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 we talk about parallelizing the algorithm, let's look at what's necessary to create the program with the IBM Parallel Environment for AIX. The example below shows the same program, but it's 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.

Note: **MPI_Comm_size** actually gets the size of the communicator you pass in and **MPI_COMM_WORLD** is a pre-defined communicator that includes everybody. For more information about these calls, *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference* or other MPI publications may be of some help. See “Encyclopedia Galactica” on page xii.

```
/*********************************************************************
*
* 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();

  return 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;
}
```

This particular example is pretty ridiculous because each parallel task is going to determine the entire inverse matrix, and they're all going to print it out. As we saw in the previous section, the output of all the tasks will be intermixed, so it will be difficult to figure out what the answer really is.

A better approach is to figure out a way to distribute the work among several parallel tasks and collect the results when they're 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's no relationship between elements of the inverse matrix, they can all be computed in parallel. Keep in mind that every communication call has an associated cost, so you need to balance the benefit of parallelism with the cost of communication. If we 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, and might outweigh the benefit of having reduced the computation cost and time by running the job in parallel. So, instead, we're going to compute the elements of each row in parallel, and send the values back, one row at a time. This way we spread some of the communication overhead over several data values. In our case, we'll 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;

  /* We need exactly one task for each row of the matrix plus one task */
  /* to act as coordinator.  If we don'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);
```

Figure 2. First Parallel Matrix Inversion Program

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, and 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's 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, we have an array of values for which we want the sine, and an outer loop would repeat this process for each element of the array. Since we don't want to recompute the factorial each time, we 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, we could assign each term to one task and just accumulate the results on a separate node. In fact, that's 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.

Note: The MPI standard, as documented in *MPI: A Message-Passing Interface Standard, Version 1.1*, available from the University of Tennesee, does not specify how the scan function is to be implemented, 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 our discussion will assume this is happening.

In our 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 x 5 to get 5!. 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 (θ) 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 Figure 2 on page 28, each task goes through the process of allocating the matrix and copying the initialization data into it. So why doesn't 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.

In order 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 in order to get the results of initialization distributed to all the other tasks. This not only means that there's nothing for the other tasks to do while one task is doing the initializing, there's 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 don't have to wait to receive the data.

So, should all initialization be done in parallel? Not necessarily. You have to keep the big picture in mind when deciding. 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's the right answer. Eventually you'll 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 whether the amount of data increases or decreases during the initialization processing. Of course, you want to transmit the smaller amount.

So, the bottom line is that 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, the IBM Parallel Environment for AIX program product interfaces with the Message Passing Subsystem, the communication software that runs on the High Performance Switch adapters (if the High Performance Switch is available and specified). The Message Passing Subsystem interfaces with a low 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 the SP system's High Performance Switch. The Message Passing Subsystem also interfaces with the IP layer.

| For optimal performance, PE uses the User Space (US) protocol as its default
| communication path. However, PE lets you run parallel applications that use the IP
| interface of the Message Passing Subsystem instead.

The User Space interface allows user applications to take full advantage of the High Performance Switch, and should be used whenever communication is a critical issue (for instance, when running a parallel application in a production environment). However, this interface cannot be used by more than one process per node at a given time (this means that only one PE session can run on a set of nodes, and only one component of the parallel program can run on each node).

The IP interface doesn't perform as well with the Message Passing Subsystem as the US interface does, but it supports multiple processes. As a result, the IP interface can be used 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.

To Thread or Not to Thread -- Protocol Implications

If you are unfamiliar with POSIX threads, don't try to learn both threads and MPI all at once. Get some experience writing and debugging serial multi-threaded programs first, then tackle parallel multi-threaded programs.

The MPI library exists in both threaded and non-threaded versions. The non-threaded library is included by the **mpcc**, **mpCC** and **mpxlf** compiler scripts. The non-threaded library is called the **signal handling** library because it uses AIX signals to ensure that messages are transmitted. The threaded library is included by the **mpcc_r**, **mpCC_r**, and **mpxlf_r** compiler scripts.

It is important to note that while a threaded program has more than one independent instruction stream, all threads share the same address space, file, and environment variables. In addition, all the threads in a threaded MPI program have the same MPI communicators, data types, ranks, and so on.

In a threaded MPI program, 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 threaded libraries and non-threaded libraries is that, in a threaded library, 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.

Note that the underlying communication subsystem provides a thread-safe message passing environment, but only one SMP processor at a time is actually copying the data between memory and the adapter.

The signal handling library has registered signal handlers for SIGALRM, SIGPIPE, (user space only), and SIGIO. The threaded library creates three service threads:

1. A thread that periodically wakes up and calls the message passing dispatcher.
2. A thread that handles interrupts generated by arriving packets.
3. A system callback thread used to implement non-blocking collective communication calls.

The service threads above are terminated when **MPI_Finalize** is called. These threads are not available to end users, except, however, the packet interrupt thread will call the signal handling function that is registered by the user to handle SIGIO, if any.

Chapter 3. Don't Panic

The *Hitchhiker's Guide to the Galaxy* revealed that 42 is what you get when you multiply 6 by 9 (which explains why things keep going wrong). Now that we all know that, we can discuss what to do in one of those situations when things go wrong. What do you do when something goes wrong with your parallel program? As the *The Hitchhiker's Guide to the Galaxy* tells us, **Don't Panic!** The IBM Parallel Environment for AIX provides a variety of ways to identify and correct problems that may arise when you're developing or executing your parallel program. This all depends on where in the process the problem occurred and what the symptoms are.

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

Note: The sample programs in this chapter are provided in full and are available from the ftp server. See “Getting the Examples in this Book” on page xiii for more information.

Before continuing, let's stop and think about the basic process of creating a parallel program. Here are the steps, (which have been greatly abbreviated):

1. Create and compile program
2. Start PE
3. Execute the program
4. Verify the output
5. Optimize the performance.

As with any process, problems can arise in any one of these steps, and different tools are required to identify, analyze and correct the problem. Knowing the right tool to use is the first step in fixing the problem. The remainder of this chapter tells you about some of the common problems you might run into, and what to do when they occur. The sections in this chapter are labeled according to the *symptom* you might be experiencing.

Messages

Message Catalog Errors

Messages are an important part of diagnosing problems, so it's essential that you not only have access to them, but that they are at the correct level. In some cases, you may get message catalog errors. This usually means that the message catalog couldn't be located or loaded. Check that your **NLSPATH** environment variable includes the path where the message catalog is located. Generally, the message catalog will be in **/usr/lib/nls/msg/C**. The value of the **NLSPATH** environment variable, including **/usr/lib/nls/msg/%L/%N** and the **LANG** environment variable, is set to *En_US*. If the message catalogs are not in the proper place, or your environment variables are not set properly, your System Administrator can probably help you. There's really no point in going on until you can read the real error messages!

© Copyright IBM Corp. 1997

The following are the IBM Parallel Environment for AIX message catalogs:

- pepoe.cat
- pempl.cat
- pepdbx.cat
- pedig.cat
- pevt.cat
- pedb.cat
- poestat.cat
- mpci_err.cat
- xprofiler.cat

Finding PE Messages

There are a number of places that you can find PE messages:

- They are displayed on the home node when it's running POE (STDERR and STDOUT).
- If you set either the **MP_PMDLOG** environment variable or the **-pmdlog** command line option to yes, they are collected in the pmd log file of each task, in **/tmp** (STDERR and STDOUT).
- They appear in the message area and pop-up windows of the parallel debugger interface.

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.***pid.taskid*, where *pid* is the process id of the Partition Manager daemon (pmd) that was started in response to the **poe** command, and *taskid* is the task number. This file contains additional diagnostic information about why the user connection wasn't made. If the file isn't there, then pmd didn't 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 for AIX: Operation and Use, Vol. 1*.

Message Format

Knowing which component a message is associated with can be helpful, especially when trying to resolve a problem. As a result, PE messages include prefixes that identify the related component. The message identifiers for the PE components are as follows.

0029-nnnn pdbx

0030-nnnn pedb

0031-nnnn Parallel Operating Environment

0031-A4nn Program Marker Array

0032-nnnn Message Passing Library

0033-1nnn Visualization Tool - Performance Monitor

0033-2nnn Visualization Tool - Trace Visualization

0033-3nnn Visualization Tool - Trace Collection

0033-4nnn Visualization Tool - Widget

2537-nnn Xprofiler X-Windows Performance Profiler

where:

- The first four digits (0029, 0030, 0031, 0032, 0033, 2537) identify the component that issued the message.
- *nnnn* identifies the sequence of the message in the group.

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

Note that you might find it helpful to run POE, the parallel debugger, or the Visualization Tool as you use this chapter.

Diagnosing Problems Using the Install Verification Program

The *Installation Verification Program* (IVP) can be a useful tool for diagnosing problems. When you installed POE, you verified that everything turned out all right 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 (both non-threaded and threaded) 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. For more information on the IVP, see Appendix C, “Installation Verification Program Summary” on page 113 or *IBM Parallel Environment for AIX: Installation Guide*.

Can't Compile a Parallel Program

Programs for the IBM Parallel Environment for AIX must be compiled with the current release of **mpxlf** or **mpcc**, etc.. If the command you're 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 the underlying compiler is installed and accessible. Your System Administrator or local AIX guru should be able to assist you in verifying these things.

Can't Start a Parallel Job

Once your program has been successfully compiled, you either invoke it directly or start the Parallel Operating Environment (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 doesn't start

  or

- POE can't connect to the remote nodes.

These problems can be caused by other problems on the home node (where you're 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. Here's what you do:

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

   ```
   $ whence poe
   ```

   If you're a C shell user, type:

   ```
   $ which poe
   ```

   If the result is just the shell prompt, you don't have POE in your path. It might mean that POE isn't installed, or that your path doesn't point to it. Check that the file **/usr/lpp/ppe.poe/bin/poe** exists and is executable, and that your PATH includes **/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 the Resource Manager is to be used to allocate remote hosts) and **MP_RMPOOL** (the pool from which Resource Manager is to allocate remote hosts) values. If they're all unset, make sure 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:

   ```
   Enter program name (or quit): _
   ```

   If you do, POE has successfully loaded, established communication with the first remote host in your host list file, validated your use of that remote host, and is ready to go to work. If you type any AIX 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:

   - Can't 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 the Resource Manager 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 IBM Parallel Environment for AIX 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.

- User Not Authorized on Remote Host

  You need an ID on the remote host and your ID on the home host (the one you are submitting the job from) must be authorized to run commands on the remote hosts. You do this by placing a **$HOME/.rhosts** file on the remote hosts that identify your home host and ID. Brush up on "Access" on page 3 if you need to. Even if you have a **$HOME/.rhosts** file, make sure 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. On the SP, this mounted file system is managed by AMD, the AutoMount Daemon. Occasionally, during user verification, the AutoMount Daemon doesn't mount your home directory fast enough, and pmd doesn't find your **.rhosts** file. Check with your System Administrator...as long as you know he doesn't bite.

  Even if the remote host is actually the same machine as your home node, you still need an entry in the **.rhosts** file. Sorry, that's the way **inetd** works.

- 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 won't give you any more information about the error, it will give you an idea of where POE was, and what it was trying to do when the error occurred. A value of 6 will give you more information than you could ever want. See Appendix A, "A Sample Program to Illustrate Messages" on page 103 for an example of the output from this setting.

Can't Execute a Parallel Program

Once POE can be started, you'll 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 5, you can use POE to invoke any AIX command or serial program on remote nodes. If you can get an AIX command or simple program, like *Hello, World!*, to run under POE, but a parallel program doesn't, 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's linked into your executable by the **mpcc** or **mpxlf** command 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 for the SP High Performance Switch. 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 variables:

```
$ export MP_EUILIB=ip
$ export MP_EUIDEVICE=en0
```

Use specific remote hosts in your host list file and don't use the Resource Manager (set **MP_RESD=no**). If you don't have a small parallel program around, recompile hello.c as follows:

```
$ mpcc -o hello_p hello.c
```

and make sure that the executable is loadable on the remote host that you are using.

Type the following command, and then look at the messages on the console:

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

If the last message that you see looks like this:

```
Calling mpci_connect
```

and there are no further messages, there's an error in opening a UDP socket on the remote host. Check to make sure that the IP address of the remote host is correct, as reported in the informational messages printed out by POE, and perform any other IP diagnostic procedures that you know of.

If you get

```
Hello, World!
```

then the communication subsystem has been successfully initialized on the one node and things ought to be looking good. Just for kicks, make sure there are two remote nodes in your host list file and try again with

```
$ poe hello_p -procs 2
```

If and when hello_p works with IP and device en0 (the Ethernet), try again with the SP High Performance Switch.

Suffice it to say that each SP node has one name that it is known by on the Ethernet LAN it is connected to and another name it is known by on the High Performance Switch. If the node name you use is not the proper name for the network device you specify, the connection will not be made. You can put the names in your host list file. Otherwise you will have to use the Resource Manager to locate the nodes.

For example,

```
$ export MP_RESD=yes
$ export MP_EUILIB=ip
$ export MP_EUIDEVICE=css0
$ poe hello_p -procs 2 -ilevel 2
```

where **css0** is the switch device name.

Look at the console lines containing the string **init_data**. These identify the IP address 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 aren't the switch IP addresses, check the Resource Manager configuration and the switch configuration.

Once IP works, and you're on an SP machine, you can try message passing using the User Space device support. User Space support is accomplished by dedicating the switch adapter on a remote host to one specific task. The Resource Manager controls which remote hosts are assigned to which users.

You can run **hello_p** with the User Space library by typing:

```
$ export MP_RESD=yes
$ export MP_EUILIB=us
$ export MP_EUIDEVICE=css0
$ poe hello_p -procs 2 -ilevel 6
```

The console log should inform you that you're using User Space support, and that the Resource Manager is allocating the nodes for you. If someone else is already running on those nodes, the Resource Manager will tell you that it can't allocate the requested nodes. You can try for other specific nodes, or you can ask the Resource Manager for non-specific nodes from a pool, but by this time, you're probably far enough along that we can just refer you to *IBM Parallel Environment for AIX: Operation and Use, Vol. 1*.

If you get a message that says POE can't load your program, and it mentions the symbol *pm_exit_value*, you are not loading POE's modification of the C run-time library. Make sure that the files **/usr/lpp/ppe.poe/lib/libc.a** and **/usr/lpp/ppe.poe/lib/libc_r.a** exist, and that the library search path (composed from MP_EUILIBPATH, MP_EUILIB, and your LIBPATH environment variable) finds these versions.

The Program Runs But...

Once you've gotten the parallel application running, it would be nice if you were guaranteed that it would run correctly. Unfortunately, this is not the case. In some cases, you may get no output at all, and your challenge is to figure out why not. In other cases, you may get output that's just not correct and, again, you must figure out why it isn't.

The Parallel Debugger is Your Friend

An important tool in analyzing your parallel program is the PE parallel debugger (**pedb** or **pdbx**). In some situations, using the parallel debugger is no different than using a debugger for a serial program. In others, however, the parallel nature of the problem introduces some subtle and not-so-subtle differences which you should understand in order to use the debugger efficiently. While debugging a serial application, you can focus your attention on the single problem area. In a parallel application, not only must you shift your attention between the various parallel tasks, you must 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, in addition to being the simplest case, it 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. Get the picture?

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 that you were not paying attention to. In these situations, your only choice may be to re-run the application and focus on the events leading up to the send.

It Core Dumps

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.

Debugging Core Dumps

There are two ways you can use 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. In our case, let's just generate it. The example we'll use is an MPI program in which even-numbered tasks pass the *answer to the meaning of life* to odd-numbered tasks. It's called **bad_life.c**, and here's 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) ==  )
        {
                char *send_message = NULL;

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

                 MPI_Recv(recv_message, 1 , 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);
}
```

We compiled **bad_life.c** with the following parameters:

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

and when we run it, we 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 generates two core files. If we list our current directory, we can indeed 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
```

So, what do we do now? Let's run **dbx** on one of the core files to see if we can find the problem. You run **dbx** like this:

```
$ dbx bad_life coredir.1/core

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

Segmentation fault in moveeq.memcpy [/usr/lpp/ppe.poe/lib/ip/libmpci.a] at 0xd055
b320
0xd055b320 (memcpy+0x10) 7ca01d2a       stsx   r5,r0,r3
(dbx)
```

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

```
(dbx) where
```

we can see the program stack:

```
moveeq._moveeq() at 0xd055b320
fmemcpy() at 0xd0568900
cpfromdev() at 0xd056791c
readdatafrompipe(??, ??, ??) at 0xd0558c08
readfrompipe() at 0xd0562564
finishread(??) at 0xd05571bc
kickpipes() at 0xd0556e64
mpci_recv() at 0xd05662cc
_mpi_recv() at 0xd050635c
MPI__Recv() at 0xd0504fe8
main(argc = 1, argv = 0x2ff22c08), line 32 in "bad_life.c"
(dbx)
```

The output of the **where** command shows that **bad_life.c** failed at line 32, so let's look at line 32, like this:

```
(dbx) func main
(dbx) list 32

   32          MPI_Recv(recv_message, 10, MPI_CHAR, taskid-1, 0,
                        MPI_COMM_WORLD, &stat);
```

When we look at line 32 of **bad_life.c**, our first guess is that one of the parameters being passed into **MPI_Recv** is bad. Let's look at some of these parameters to see if we can find the source of the error:

```
(dbx) print recv_message
(nil)
```

Ah ha! Our receive buffer has not been initialized and is NULL. The sample programs for this book include a solution called **good_life.c**. See “Getting the Examples in this Book” on page xiii for information on how to get the sample programs.

It's important to note that we compiled **bad_life.c** with the **-g** compile flag. This gives us all the debugging information we need in order to view the entire program state and to print program variables. In many cases, people don't compile their programs with the **-g** flag, and they may even turn optimization on (**-O**), so there's virtually no information to tell them what happened when their 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.

In the following example, we'll use **bad_life** again, but we'll use **pdbx** instead of **dbx**. Load **bad_life.c** under **pdbx** with the following command:

```
$ pdbx bad_life

pdbx Version 2.1 -- Apr 30 1996 15:56:32

  0:reading symbolic information ...
  1:reading symbolic information ...
  2:reading symbolic information ...
  3:reading symbolic information ...
  1:[1] stopped in main at line 12
  1:   12       char            *send_message = NULL;
  0:[1] stopped in main at line 12
  0:   12       char            *send_message = NULL;
  3:[1] stopped in main at line 12
  3:   12       char            *send_message = NULL;
  2:[1] stopped in main at line 12
  2:   12       char            *send_message = NULL;
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.
  3:
  3:Segmentation fault in @moveeq._moveeq [/usr/lpp/ppe.poe/lib/ip/libmpci.a]
  at 0xd036c320
  3:0xd036c320 (memmove+0x10) 7ca01d2a       stsx   r5,r0,r3
  1:
  1:Segmentation fault in @moveeq._moveeq [/usr/lpp/ppe.poe/lib/ip/libmpci.a]
  at 0xd055b320
  1:0xd055b320 (memcpy+0x10) 7ca01d2a       stsx   r5,r0,r3
```

Once we get segmentation faults, we can focus our attention on one of the tasks that failed. Let's look at task 1:

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

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

```
pdbx(1) where

  1:@moveeq.memcpy() at 0xd055b320
  1:fmemcpy() at 0xd0568900
  1:cpfromdev() at 0xd056791c
  1:readdatafrompipe(??, ??, ??) at 0xd0558c08
  1:readfrompipe() at 0xd0562564
  1:finishread(??) at 0xd05571bc
  1:kickpipes() at 0xd0556e50
  1:mpci_recv() at 0xd05662fc
  1:_mpi_recv() at 0xd050635c
  1:MPI__Recv() at 0xd0504fe8
  1:main(argc = 1, argv = 0x2ff22bf0), line 32 in "bad_life.c"
```

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

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

Next, we'll list line 32, which is where the problem is located:

```
pdbx(1) l 32

  1:   32        MPI_Recv(recv_message, 10, MPI_CHAR, taskid-1, 0,
                 MPI_COMM_WORLD, &stat);
```

Now that we're at line 32, we'll print the value of **recv_message**:

```
pdbx(1) p recv_message

  1:(nil)
```

As we can see, our program passes a bad parameter to **MPI_RECV()**.

Both the techniques we've talked about so far help you find the location of the problem in your code. The example we used makes it look easy, but in many cases it won't be so simple. However, knowing where the problem occurred is

valuable information if you're forced to debug the problem interactively, so it's worth the time and trouble to figure it out.

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, so it is of 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, if possible, use the attach capability of **dbx**, **xldb**, **pdbx** or **pedb** to examine the task while it's still running.

No Output at All

Should There Be Output?

If you're getting no output from your program and you think you ought to be, the first thing you should do is 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 won't 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). 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's important to check the messages carefully.

Figuring Out Return Codes: It's 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, but if 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, more analysis is required.

It Hangs

If you've gotten this far and the POE messages and the additional checking by the message passing routines have been unable to 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 Try Using the Visualization Tool

One way to do this is to run with Visualization Tool (**VT**) tracing enabled, and examine the tracefile. To do this, compile your program with the **-g** flag and run the program with the **-tracelevel 9** command line option, or by setting the **MP_TRACELEVEL** environment variable to 9.

When your program terminates (either on its own or via a <**Ctrl-c**> from you), you will be left with a file called **pgmname.trc** in the directory from which you submitted the parallel job. You can view this file with **VT**.

Let's look at the following example...it's got 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 we're not processing the pixel, all we 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 we aren't doing anything with the pixel data, we 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 was taken 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, we're just 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 doesn't 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 we're using the **-g** option when we compile the example. We're cheating a little because we know there's going to be a problem, so we're compiling with debug information 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, we suspect the program is hanging somewhere. We know it starts executing because we get some messages from it. It could be a logical hang or it could be a communication hang. There are two ways you can approach this problem; by using either VT or the attach feature of **pedb** (the PE parallel debugger). We'll start by describing how to use VT.

Because we don't know what the problem is, we'll turn on full tracing with **-tracelevel 9**, as in the example below.

```
$ rtrace_bug -procs 4 -labelio yes -tracelevel 9
  1:Compute #1: checking in
  3:Compute #3: checking in
  2:Compute #2: checking in
  0:Control #0: No. of nodes used is 4
  0:Control: expect to receive 2500 messages
  2:Compute #2: done sending. Task 2 waiting to complete.
  1:Compute #1: done sending. Task 1 waiting to complete.
  3:Compute #3: done sending. Task 3 waiting to complete.
^C
ERROR: 0031-250  task 0: Interrupt
ERROR: 0031-250  task 3: Interrupt
ERROR: 0031-250  task 1: Interrupt
ERROR: 0031-250  task 2: Interrupt
$ ls -alt rtrace_bug.trc
-rw-r--r--   1 vt       staff     839440 May  9 14:54 rtrace1.trc
```

When you run this example, make sure you press <**Ctrl-c**> as soon as the last *waiting to complete* message is shown. Otherwise, the trace file will continue to grow with kernel information, and it will make visualization more cumbersome. This will create the trace file in the current directory with the name **rtrace_bug.trc**.

Now we can start VT with the command:

```
$ vt -tracefile rtrace_bug.trc
```

VT will start with the two screens shown in Figure 3 and Figure 4 on page 52 below. One is the VT Control Panel:

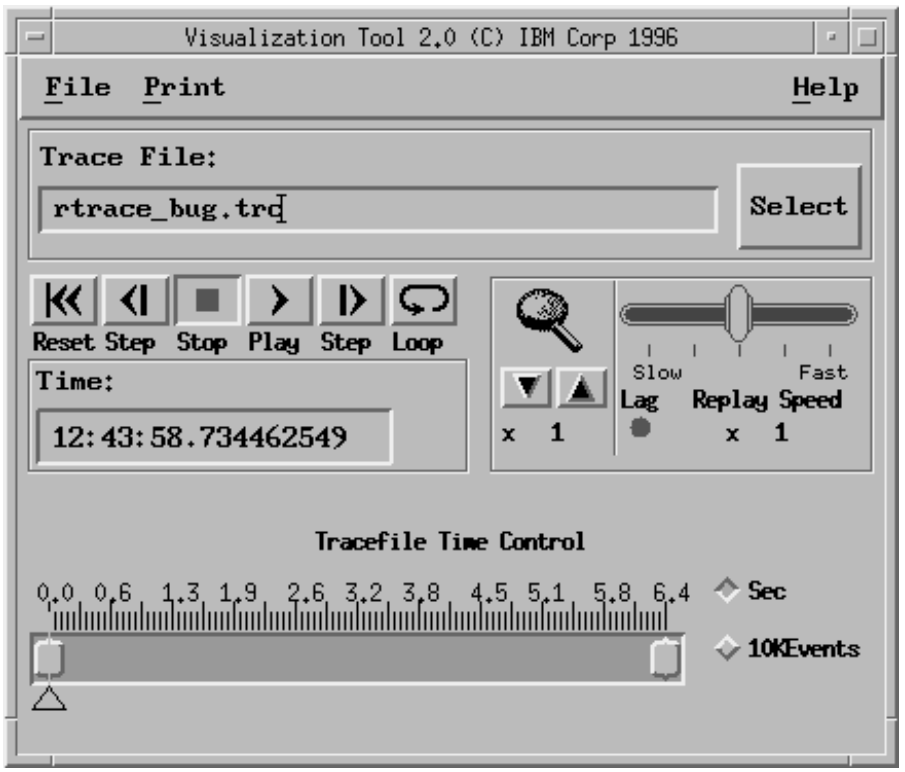

Figure 3. VT Control Panel

The other is the VT View Selector:

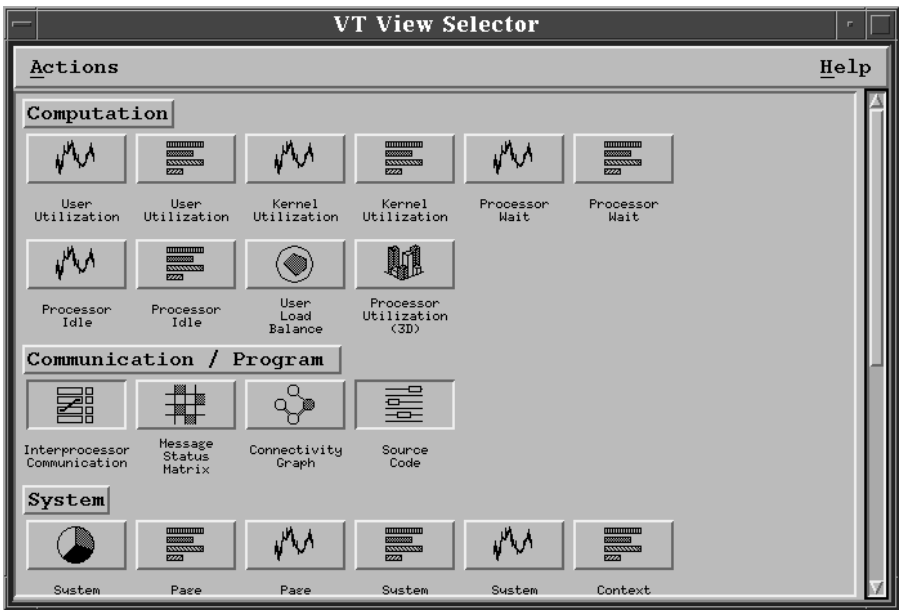

Figure 4. VT View Selector

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 haven't been released are common, in addition to the deadlock possibilities that arise from improper use of the MPI message passing calls.

Using the VT Displays

Since we don't know exactly what the problem is, we should look at communication and kernel activity. Once VT has started, you should select the Interprocessor Communication, Source Code, and System Summary displays from the *View Selector* panel (Figure 4). To do this, go to the VT View Selector Panel. Locate the category labeled *Communication/Program*. Beneath that category label is a toggle button labeled *Interprocessor Communication*.

PLACE the mouse cursor on the toggle button.

PRESS the left mouse button.

▲ The Interprocessor Communication display appears.

Next, locate the toggle button in the same category labeled *Source Code* and select it in a similar manner. Finally, locate the *System Summary* toggle button in the *System* category and select it. You may need to resize the System Summary to make all four pie charts visible.

If you experience any problems with the Source Code display, it may be because **rtrace_bug** was not compiled with the **-g** option, or because the source and executable do not reside in the directory from which you started VT. Although, you can use the **-spath** command line option of VT to specify paths for locating the source code, for this example you should make sure that the source, executable, and trace file all exist in the directory from which you are starting VT.

Using the VT Trace File to Locate Communication Problems

Since all we know is that the program appeared to *hang*, the best bet at this point is to just play the trace file and watch for activity that appears to cease. Place the mouse cursor over the Play button on the main control panel and press the left mouse button to begin playback. This may take some time as VT attempts to play the trace file back at the same speed it was captured. If you want to speed up playback, you can use the mouse to drag the *Replay Speed* slider to the right.

You're looking for one of the following things to happen:

- One of the processes will go into a communication state on the Interprocessor Communication display and remain there for the duration of the trace file.

  or

- The Source Code display will show the processes stopped at a communication call, but the Interprocessor Communication display will show the processes not in a communication state.

  or

- The Interprocessor Communication display will show fewer communication events than you expected to see (possibly none at all!).

In the first case, a blocking communication call (for example, a Receive) has been invoked by a process, but the call was never satisfied. This can happen if a Blocking Receive, or a Wait on a Non-Blocking Receive is issued, and the corresponding Send never occurs. In this case, the Source Code display can show you where in the source code the blocked call occurred.

If the Interprocessor Communication display shows that the processes are not in a communication state, and the Source Code display shows no activity for one or more processes, then the code is probably in a standard infinite loop. You can deduce this because the Source Code display only tracks message passing calls. Once the display shows a process at a communication call, the position of that process will not change until another communication call is performed by that process. If the Interprocessor Communication display shows that the process is no longer in the communication call, you can conclude that the process has completed the call but has not reached another call. If you expected the process to go on to the next call, something in the non-communication code has prevented it from doing so. In many cases, this will be an infinite loop or blocking in I/O (although there are other legitimate causes as well, such as blocking while waiting on a signal).

If the trace file shows less or different communication activity than you expected, you should use the Reset button on the VT control panel to start the trace file from the beginning and then step the trace file, one communication event at a time, until you determine where the execution does something you didn't expect.

In each of these scenarios, you can use the VT Source Code display to locate the area that appears to contain the problem, and use that knowledge to narrow the focus of your debugging session.

Another cause for no activity in the trace file for a process, is failure of the network connection to the node during execution. Normally, POE checks to make sure its connections to all remote hosts are healthy by periodic message exchange (we call it the POE pulse). If the network connection fails, the POE pulse will fail and POE

will time out, terminating the job. You can turn the pulse off by setting the **MP_PULSE** environment variable to 0, but then you are responsible for detecting if a remote host connection fails.

In our example, the trace file playback ends with the Interprocessor Communication display and the Source Code display appearing as they do in Figure 5 and Figure 6 on page 55.

The Interprocessor Communication display, below, shows that process 0 is in a blocking receive. You can tell this by placing the mouse cursor over the box that has the zero in it and pressing the left mouse button. The pop-up box that appears will tell you so. The other three processes are in an **MPI_Barrier** state. What this means is that process 0 is expecting more input, but the other processes are not sending any.

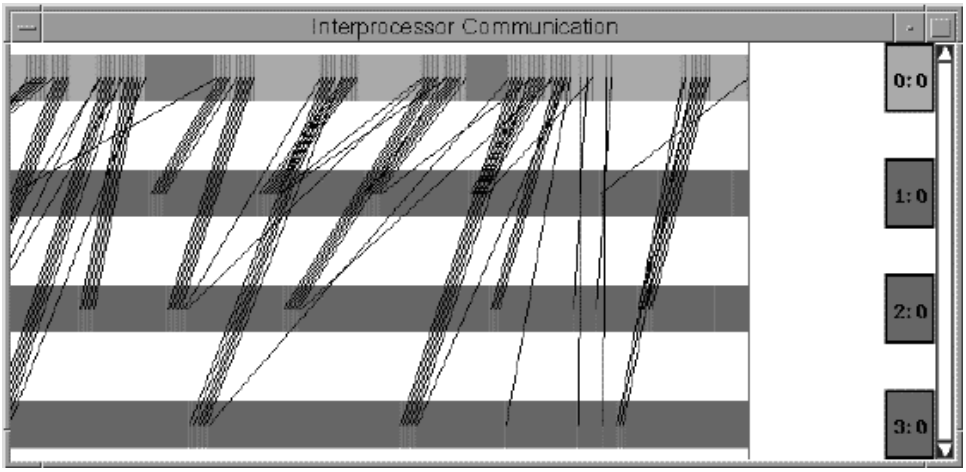

Figure 5. Interprocessor Communication Display

The Source Code display, below, shows that Process 0 is at the **MPI_Recv** call in **collect_pixels**. Process 0 will not leave this loop until the variable **mx** becomes 0. The other processes are in the **MPI_Barrier** call that they make after **compute_pixels** finishes. They have sent all the data they think they should, but Process 0 is expecting more.

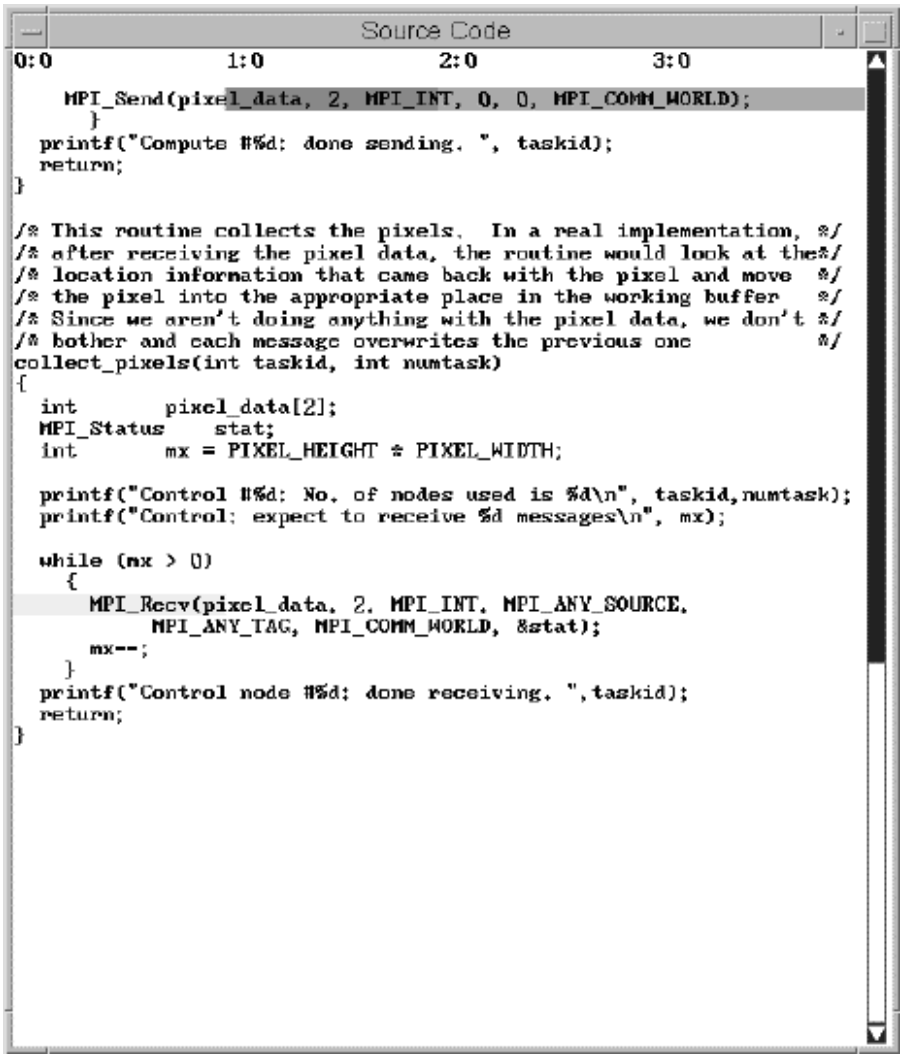

Figure 6. Source Code Display

Let's Attach the Debugger

After using VT, is seems to be clear that our program is hanging. Let's use the debugger to find out why. The best way to diagnose this problem is to attach the debugger directly to our POE job.

Start up POE and run **rtrace_bug**:

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

To attach the debugger, we first need to get the process id (pid) of the POE job. You can do this with the AIX **ps** command:

```
$ ps -ef | grep poe

smith 24152 20728   0 08:25:22  pts/0  0:00 poe
```

Next, we'll need to start the debugger in attach mode. Note that we can use either the **pdbx** or the **pedb** debugger. In this next example, we'll use **pedb**, which we'll start it in attach mode by using the **-a** flag and the process identifier (pid) of the POE job:

```
$ pedb -a 24152
```

After starting the debugger in attach mode, the **pedb** Attach Dialog window appears:

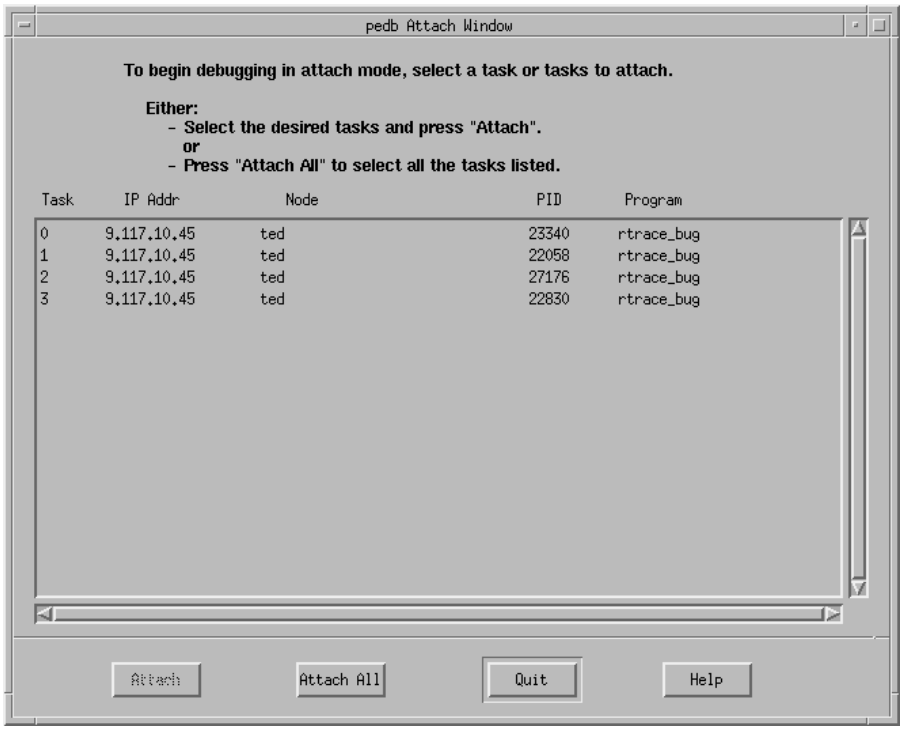

Figure 7. Attach Dialog Window

The Attach Dialog Window contains a list of task numbers and other information that describes the POE application. It provides information for each task in the following fields:

Task The task number

IP The IP address of the node on which the task or application is running

Node The name, if available, of the node on which the task or application is running

PID The process identifier of the selected task

Program The name of the application and arguments, if any. These may be different if your program is MPMD.

At the bottom of the window there are two buttons (other than Quit and Help):

Attach Causes the debugger to attach to the tasks that you selected. This button remains grayed out until you make a selection.

Attach All Causes the debugger to attach to **all** the tasks listed in the window. You don't have to select any specific tasks.

Next, select the tasks to which you want to attach. You can either select all the tasks by pressing the **Attach All** button, or you can select individual tasks, by pressing the **Attach** button. In our example, since we don't know which task or set

of tasks is causing the problem, we'll attach to all the tasks by pressing the **Attach All** button.

PLACE the mouse cursor on the Attach All button.

PRESS the left mouse button.

▲ The Attach Dialog window closes and the debugger main window appears:

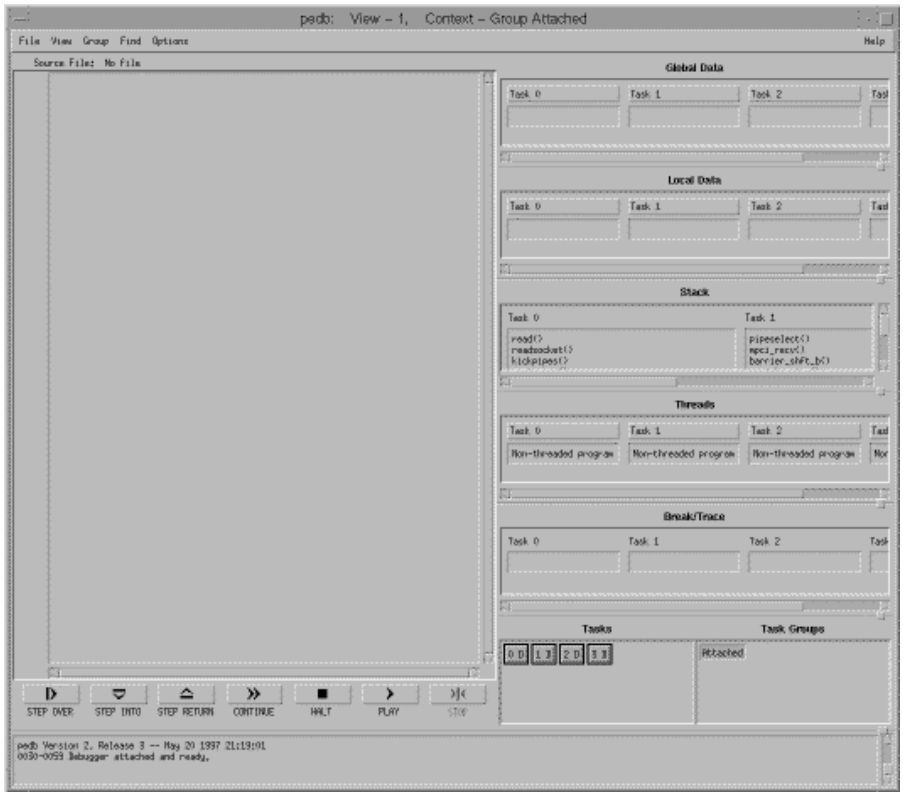

Figure 8. pedb Main Window

Since our code is hung in low level routines, the initial main window only provides stack traces. To get additional information for a particular task, double click on the highest line in the stack trace that has a line number and a file name associated with it. This indicates that the source code is available.

For task 0, in our example, this line in the stack trace is:

```
collect_pixels(), line 101 in rtrace_bug.c
```

Clicking on this line causes the local data to appear in the *Local Data* area and the source file (the **compute_pixels** function) to appear in the *Source File* area. In the source for **compute_pixels**, line 101 is highlighted. Note that the function name and line number **within** the function your program last executed appears here. (in this case, it was function **MPI_Recv**() on line 101).

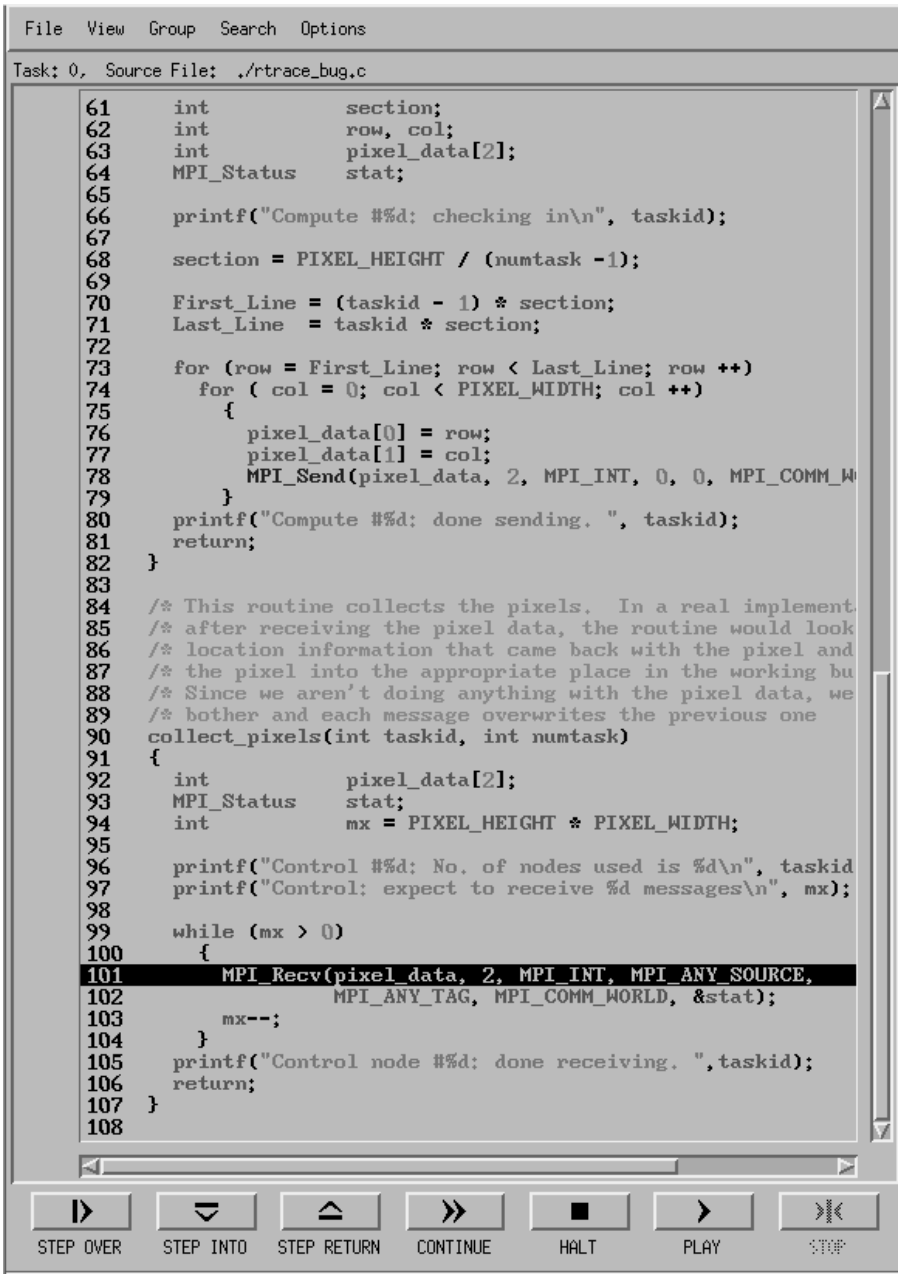

Figure 9. Getting Additional Information About a Task

PLACE the mouse cursor on the Task 0 label (not the box) in the Global Data area.

PRESS the right mouse button.

▲ a pop-up menu appears.

SELECT the Show All option.

▲ All the global variables for this are tracked and displayed in the window below the task button.

Repeat the steps above for each task.

Now you can see that task 0 is stopped on an **MPI_Recv()** call. When we look at the Local Data values, we find that **mx** is still set to 100, so task 0 thinks it's still going to receive 100 messages. Now, lets look at what the other tasks are doing.

To get information on task 1, go to its stack window and double click on the highest entry that includes a line number. In our example, this line is:

```
main(argc = 1, argv = 0x2ff22a74), line 43, in rtrace_bug.c
```

Task 1 has reached an **MPI_Barrier()** call. If we quickly check the other tasks, we see that they have all reached this point as well. So....the problem is solved. Tasks 1 through 3 have completed sending messages, but task 0 is still expecting to receive more. Task 0 was expecting 2500 messages but only got 2400, so it's still waiting for 100 messages. Let's see how many messages each of the other tasks are sending. To do this, we'll look at the global variables **First_Line** and **Last_Line**. We can get the values of **First_Line** and **Last_Line** for each task by selecting them in the Global Data area.

PLACE the mouse cursor over the desired task number label (not the box) in the Global Data area.

PRESS the right mouse button.

▲ a pop-up menu appears.

SELECT the Show All option.

▲ The **First_Line** and **Last_Line** variables are tracked and displayed in the window below the task button.

Repeat the steps above for each task.

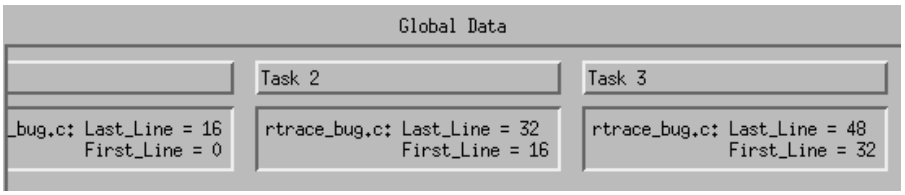

Figure 10. Global Data Window

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 we're missing 2 rows, that explains the 100 missing messages. As you've probably already figured out, the division of the total number of lines by the number of tasks is not integral, so we lose part of the result when it's converted back to an integer. Where each task is supposed to be processing 16 and two-thirds lines, it's only handling 16.

Fix the Problem

So how do we fix this problem permanently? As we mentioned above, there are many ways:

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

In our case, since Task 1 was responsible for 16 and two thirds rows, it would process rows 0 through 16. Task 2 would process 17-33 and Task 3 would process 34-49. The way we're going 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. You can access the entire program via anonymous ftp. See “Getting the Examples in this Book” on page xiii for more information.

```
/*****************************************************************************
*
* 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.)

What's the Hangup?

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. The Visualization Tool (VT), of the IBM Parallel Environment for AIX, can show you what's going on in the program when the hang occurs, and this information can be used with the parallel debugger to identify the specific cause of the hang.

However, sometimes analysis under the debugger indicates that the source of a hang is a message that was never received, even though it's 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.

Although MPI specifies many details about the interface and behavior of communication calls, it also leaves many implementation details unspecified (and it doesn't 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 Appendix B, “MPI Safety” on page 107.

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:

```
$ 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:

```
$ no -a
```

The value presented for **thewall** is in K Bytes. You can use the **no** command to change this value. For example,

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

sets **thewall** to 16 M Bytes.

Message passing between lots of remote hosts can tax the underlying IP system. Make sure 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'll need to get your system support people involved to isolate the problem.

Other Hangups

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 for AIX: Operation and Use, Vol. 1* for information on the **poekill** command.

There are situations where POE itself can hang. Usually these 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; allocate lots of **mbufs** on the home node, and don't 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

The causes of error messages are tracked down and corrected 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's 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

Bad results are tracked down and corrected in a parallel program in a fashion similar to that used for serial programs. The process, as we saw 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, so it's very difficult to correlate a message that you receive with a particular series of events.

Debugging and Threads

So far, we've talked about debugging normal old serial or parallel programs, but you may want to debug a threaded program (or a program that uses threaded libraries). 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 you're working in. You have the potential to create a multi-threaded application, using a multi-threaded 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're 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 with 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.

If you're using the threaded MPI library, the 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.

It's important to note that to effectively debug your application you must be aware of how threads are dispatched. When a task is stopped, all threads are also 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 haven't 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 sender 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...the window of opportunity for the MPI threads to process the message is brief, and is only open 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 tools (**pdbx** and **pedb**), see IBM Parallel Environment for AIX: Operation and Use, Vol. 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 for AIX: MPI Programming and Subroutine Reference.

Keeping an Eye on Progress

Often, once a program is running correctly, you'd like to keep tabs on its progress. Frequently, in a sequential program, this is done by printing to standard output. However, if you remember from Chapter 1, standard output from all the tasks is interleaved, and it is difficult to follow the progress of just one task. If you set the **MP_STDOUTMODE** environment variable to **ordered**, you can't see how the progress of one task relates to another. In addition, normal output is not a blocking operation. This means that a task that writes a message will continue processing so that by the time you see the message, the task is well beyond that point. This makes it difficult to understand the true state of the parallel application, and it's especially difficult to correlate the states of two tasks from their progress messages. One way to synchronize the state of a parallel task with its output messages is to use the *Program Marker Array* (**pmarray**).

Note: If you are unfamiliar with the Program Marker Array, you may find it helpful to refer to IBM Parallel Environment for AIX: Operation and Use, Vol. 1, for more information.

The *Program Marker Array* consists of two components: the display function, **pmarray**, and the instrumentation call, **mpc_marker**. When **pmarray** is running, it shows a display that looks like Figure 11, below.

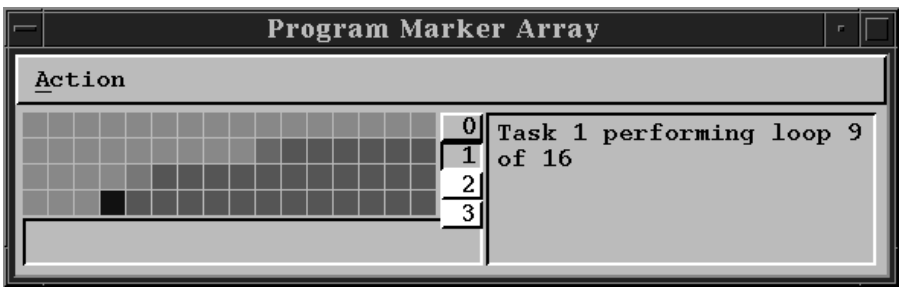

Figure 11. Program Marker Array

Each row of colored squares is associated with one task, which can change the color of any of the lights in its row with the **mpc_marker** call. The declaration looks like this in FORTRAN:

```
MP_MARKER (INTEGER LIGHT, INTEGER COLOR, CHARACTER STRING)
```

And it looks like this in C:

```
void mpc_marker(int light, int color, char *str)
```

This call accepts two integer values and a character string. The first parameter, **light**, controls which light in the **pmarray** is being modified. You can have up to 100 lights for each task. The second parameter, **color**, specifies the color to which you are setting the light. There are 100 colors available. The third parameter is a string of up to 80 characters that is a message shown in the text area of the **pmarray** display.

Before you start the parallel application, you need to tell **pmarray** how many lights to use, as well as how many tasks there will be. You do this with the **MP_PMLIGHTS** and the **MP_PROCS** environment variables.

```
$ export MP_PROCS=4
$ export MP_PMLIGHTS=16
```

If the parallel application is started from an *X-Windows* environment where **pmarray** is running, the output square of **pmarray**, for the task that made the call in the position specified by the **light** parameter, changes to the color specified by the **color** parameter. The character string is displayed in a text output region for the task. In addition to providing a quick graphical representation of the progress of the application, the output to **pmarray** is synchronized with the task that generates it. The task will not proceed until it has been informed that the data has been sent to **pmarray**. This gives you a much more current view of the state of each task.

The example below shows how **pmarray** can be used to track the progress of an application. This program doesn't do anything useful, but there's an inner loop that's executed 16 times, and an outer loop that is executed based on an input parameter. On each pass through the inner loop, the **mpc_marker** call is made to color each square in the task's **pmarray** row according to the color of the index for the outer loop. On the first pass through the inner loop, each of the 16 squares will be colored with color 0. On the second pass, they will be colored with color 1. On each pass through the outer loop, the task will be delayed by the number of seconds equal to its task number. Thus, task 0 will quickly finish but task 4 will take a while to finish. The color of the squares for a task indicate how far they are through the outer loop. The square that is actually changing color is the position in the inner loop. In addition, a text message is updated on each pass through the outer loop.

```
/*****************************************************************************
*
* Demonstration of use of pmarray
*
* To compile:
* mpcc -g -o use_pmarray use_pmarray.c
*
*****************************************************************************/

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

int main(int argc, char **argv)
{
  int i, j;
  int inner_loops = 16, outer_loops;
  int me;
  char buffer[256];
```

```
  time_t start, now;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&me);

  if(argc>1) outer_loops = atoi(argv[1]);
  if(outer_loops<1) outer_loops = 16;

  for(i=0;i<outer_loops;i++)
    {
      /* Create message that will be shown in pmarray text area */
      sprintf(buffer,"Task %d performing loop %d of %d",me,i,outer_loops);
      printf("%s\n",buffer);
      for(j=0;j<inner_loops;j++)
        {
          /* pmarray light shows which outer loop we are in  */
          /* color of light shows which inner loop we are in */
          /* text in buffer is created in outer loop         */
          mpc_marker(i,5*j,buffer);
        }

      /* Pause for a number of seconds determined by which */
      /* task this is.  sleep(me) cannot be used because   */
      /* underlying communication mechanism uses a regular */
      /* timer interrupt that interrupts the sleep call    */
      /* Instead, we'll look at the time we start waiting  */
      /* and then loop until the difference between the    */
      /* time we started and the current time is equal to  */
      /* the task id                                       */
      time(&start);
      time(&now);
      while(difftime(now,start)<(double)me)
         {
           time(&now);
         }
    }
  MPI_Finalize();
  return 0;
}
```

Before running this example, you need to start **pmarray**, telling it how many lights to use. You do this with the **MP_PMLIGHTS** environment variable.

In our example, if we wanted to run the program with eight outer loops, we would set **MP_PMLIGHTS** to **8** before running the program.

Although it's not as freeform as print statements, or as extensible, **pmarray** allows you to send three pieces of information (the light number, the color, and the text string) back to the home node for presentation. It also ensures that the presentation is synchronized as closely to the task state as possible. We recommend that if you use **pmarray** for debugging, you define a consistent strategy for your application and stick with it. For example, you may want to use color to indicate state (initializing, active, disabled, terminated), and light number to indicate module or subsystem. You can configure **pmarray** with as many lights as will fit on the display.

Chapter 4. So Long And Thanks For All The Fish

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

In general, performance tuning can be a complicated task. *The Hitchhiker's Guide to the Galaxy* tells us that dolphins were the most intelligent life form on the Earth. When the Earth was being destroyed to make way for an Interstellar Bypass, it seems fitting that the dolphins were the only species to leave the planet. Their parting words, to a few select humans, were “So long and thanks for all the fish.” The people that tune the performance of applications regard the original developers of their code in much the same way (even if they, themselves, were the original developers); they appreciate the working code, but now they've got more complex things to do, that are significantly different than code development.

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.

Note: It may not be possible to use some tools in a parallel environment in the same way that they're 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.

A report from Cornell University found that the performance results obtained by these techniques yielded comparable results. The difference was in the tools that were used in each of the approaches, and how they were used.

With either approach, you use the standard sequential tools in the traditional manner. When an application is tuned and then parallelized, you need to 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 the workload needs to be rebalanced. Or if a task starts waiting for a message long

© Copyright IBM Corp. 1997

before it arrives, perhaps it could do more algorithmic processing before waiting for the message. When an application is parallelized 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 section will not deal with standard algorithmic tuning techniques. Rather, we will discuss some of the ways the PE can help you tune the parallel nature of your application, regardless of the approach you take. To illustrate this, we'll use two examples.

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'd like perfect balance among the tasks but doing so may require additional communication that actually makes the performance worse. We discussed this briefly in “Duplication Versus Redundancy” on page 32 when we said that sometimes it's 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, “The Answer is 42” on page 21. We showed you how to start making the sequential program into a parallel one by distributing the element calculation once the determinant was found. What we didn't show you was how poor a start this actually is. Part of the program is shown below. You can access the complete program via anonymous ftp. See “Getting the Examples in this Book” on page xiii for more information.

```
************************************************************************
*
* 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;
}
```

To illustrate the problem, run the parallel matrix inversion program, **inverse_parallel.c** with tracing turned on, and then run VT on the resulting trace file.

```
$ mpcc -g -o inverse_parallel inverse_parallel.c
$ inverse_parallel -tracelevel 9 -procs 9
$ vt -tracefile inverse_parallel.trc
```

Select the *Interprocessor Communication*, *Source Code*, *Message Matrix* and the *System Summary* displays and begin playing the trace file. You'll see a significant amount of CPU activity, both kernel and user, with very little idle of wait time on the *System Summary* display but no communication activity for a while. Finally, you'll see messages send back to the last task as each server sends its results back. So what's going on? Because VT only records source code location for message library calls, we can't use it to locate where the time is being spent. However, we can go back to **gprof**.

Recompile **inverse_parallel.c** with the **-pg** flag and re-run it. Run **gprof** on the monitor files for tasks 0-7 (we know task 8 just collects the results so we aren't that concerned with its performance).

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

  %   cumulative   self              self     total
 time   seconds  seconds     calls  ms/call  ms/call  name
 80.3       9.81     9.81       72   136.25   136.25  .determinant [1]
  8.8      10.89     1.08                             .__mcount [5]
  3.6      11.33     0.44                             .kickpipes [6]
  0.9      11.44     0.11                             .readsocket [7]
  0.0      12.21     0.00       64     0.00   136.25  .coefficient [4]
  0.0      12.21     0.00        8     0.00  1226.25  .main [2]
```

We see that we 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 matrices formed from the last two rows and each combination of columns. This isn't 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 matrices. If the 2 by 2 matrix determinants can be captured and re-used, 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 matrices 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. You can access the complete program via anonymous ftp. See “Getting the Examples in this Book” on page xiii for more information.

Here's 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's the partial determinant call:

```
}

/* Compute the determinants of all 2x2 matrices created by combinations */
/* of columns of the bottom 2 rows                                      */
/* partial_determinant[i] points to the first determinant of all the 2x2*/
/* matricies formed by combinations with column i.  There are n-i-1     */
/* such matricies (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;
}
```

Now when we look at the VT trace for this version of the program, we can see the initial communication that occurs as the tasks cooperate in computing the partial determinants. 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 your program.

- Two environment variables affect the overhead of an MPI call in the threaded library:
  - MP_SINGLE_THREAD=[NO|YES]
  - MP_EUIDEVELOP=[NO|YES|DEB|NOC]

  A program that has only one MPI communication thread may set the environment variable MP_SINGLE_THREAD=YES before calling MPI_Init. 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.

The MP_EUIDEVELOP environment variable lets you control how much checking is done when you run your program. Eliminating checking altogether (setting MP_EUIDEVELOP to NO) provides performance (latency) benefits, but may cause critical information to be unavailable if your executable hangs due to message passing errors. For more information on MP_EUIDEVELOP and other POE environment variables, see *IBM Parallel Environment for AIX: Operation and Use, Vol. 1*.

- Generally, on a uniprocessor, one should expect the performance of the signal library to be somewhat better than the thread library, since the thread library must make additional checks to insure thread safety (even if the user has only one communication thread). On an SMP, it may be possible to realize overlap of computation and communication using the thread library that is difficult if not impossible to achieve with the signal library. In that case, a task may run faster with the threaded library.
- 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 only contain a summary of the information from all the threads per task together. Viewing the data using gprof or Xprofiler is limited to only showing this summarized data on a per task basis, not per thread.

For more information on profiling, see *IBM Parallel Environment for AIX: Operation and Use, Vol. 2*.

For more information on MPI tracing and visualization using VT, see *IBM Parallel Environment for AIX: Operation and Use, Vol. 2*. Support is included for thread safe tracing within the threaded MPI library and limited visualization using VT.

Why is this so slow?

So, you've got a serial program and you want it to execute faster. In this situation, it's best not to jump into parallelizing your program right away. Instead, you start by tuning your serial algorithm.

The program we'll use in this next example approximates the two-dimensional Laplace equation and, in our case, uses a 4-point stencil.

Our algorithm is very straight-forward; for each array element, we'll 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).

Note: You may find it helpful to refer to *In Search of Clusters* by Gregory Phister for more information on this problem and how to parallelize it. See “Related Non-IBM Publications” on page xiii.

Note that the 4-point stencil program is central to this entire section, so you may want to spend some time to understand how it works.

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

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

Next, we 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
```

The following table shows the result:

Program Name	Tasks	Wallclock Time	Array Size per Task
naive	1 (single processor)	11m1.94s	1000x1000

Note: The figures in the table above, as well as the others in this section, provide results that were gathered on an IBM RS/6000 SP. Your execution time may vary, depending on the system you're using.

Looking at these results, we can see that there's some room for improvement, especially if we scale the problem to a much larger array. So, how can we improve the performance?

Profile it

The first step in tuning our program is to find the areas within it that execute most of the work. Locating these compute-intensive areas within our program lets us focus on the areas that give us the most benefit from tuning. How do we find these areas? The best way to find them is to *profile* your program.

When we profile our program, we need to compile it with the **-pg** flag to generate profiling data, like this:

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

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

Now that we've compiled our program with the **-pg** flag, let's run it again to see what we get:

```
$ naive
```

This generates a file called **gmon.out** in the current working directory. We can look at the contents of **gmon.out** with the Xprofiler profiling tool. To start Xprofiler, we'll use the **xprofiler** command, like this:

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

The Xprofiler main window appears, and in this window you'll see the **function call tree**. The function call tree is a graphical representation of the functions within your application and their inter-relationships. 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 called *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 your application appear within a fenced-in area called a *cluster box*. For the purposes of this section, we'll remove the cluster boxes from the display.

For more information on Xprofiler, see IBM Parallel Environment for AIX: Operation and Use, Vol. 2.

PLACE the mouse cursor over the Filter menu.

CLICK the left mouse button.

▲ The Filter menu appears.

SELECT the **Remove 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. We can get the name of the function by looking a little more closely at it:

PLACE the mouse cursor over the View menu.

CLICK the left mouse button.

▲ The View menu appears.

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

CLICK the left mouse button.

▲ The Overview Window appears.

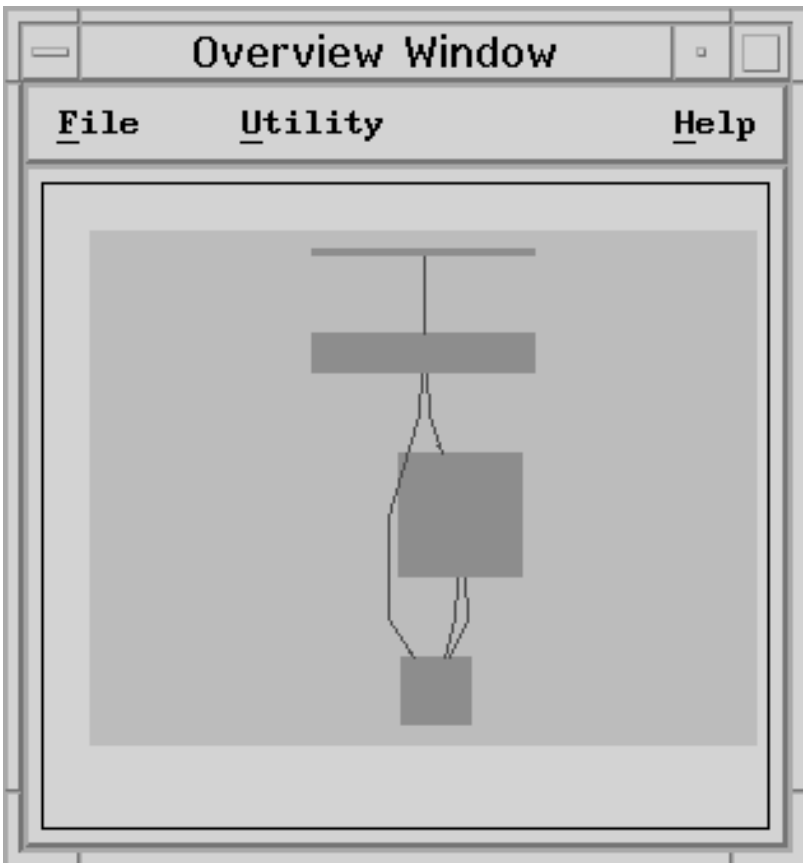

Figure 12. Xprofiler overview window

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 your 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's 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 wasn't within the highlight area, it it didn't get magnified. If this was the case, you'll need to move the highlight area:

PLACE the mouse cursor anywhere within the highlight area.

PRESS and HOLD the left mouse button.

DRAG the highlight, 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 we can now see that most of the work is being done in the **compute_stencil()** subroutine, so this is where we should focus our attention.

It's important to note that the programming style you choose can influence your 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 your 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. See “Related IBM Publications” on page xiii.

As far as we're concerned, the goal here is not to use every optimization trick that's ever been thought up. Instead, we should focus on some basic techniques that can produce the biggest performance boost for the time and effort we spend.

The best way to begin is to look at our use of memory (including hardware data cache) as well as what we're doing in the critical section of our code. So.....let's look at our 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, we can see that our compute subroutine is traversing our 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 we should make is to reorder our loops so that they traverse down columns rather than across rows. This should provide a reasonable performance boost. Note that it's 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 don't change when the loops are reordered, then it's safe to change their order. In the example we just looked at, it was safe to reorder the loops, so here's what the revised program looks like. Notice that all we did was swap the order of the loops.

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

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

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

As before, we need to time our run, like this:

```
$ time reordered
```

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

Program Name	Tasks	Wallclock Time	Array Size per Task
naive	1 (single processor)	11m1.94s	1000x1000
reordered	1 (single processor)	5m35.38s	1000x1000

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

Parallelize it

Now that we feel confident that our serial program is reasonably efficient, we should look at ways to parallelize it. As we discussed in Chapter 2, “The Answer is 42” on page 21, there are many ways to parallelize a program, but the two most commonly used techniques are functional decomposition and data decomposition. We'll focus on data decomposition only.

OK, so how do I *decompose* my data? Let's start by dividing the work across the processors. Each task will compute a section of an array, and each program will solve $\frac{1}{n}$ of the problem when using *n* processors.

Here's the algorithm we'll use:

- 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 our 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 our 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
```

Program Name	Tasks	Wallclock Time	Array Size per Task
naive	1 (single processor)	11m1.94s	1000x1000
reordered	1 (single processor)	5m35.38s	1000x1000
chaotic	4 (processors)	2m4.58s	500x500

The results above show that we more than doubled performance by parallelizing our program. So...we're done, right? Wrong! Since we divided up the work between four processors, we expected our program to execute four times faster. Why doesn't it? This could be due to one of several factors that tend to influence overall performance:

- Message passing overhead
- Load imbalance
- Convergence rates

We'll look at some of these factors later. Right now we need to be asking ourselves something more important; does the parallel program get the same answer?

The algorithm we chose gives us a *correct* answer, but as you will see, it doesn't give us the *same* answer as our serial version. In practical applications, this may

be OK. In fact, it's very common for this to be acceptable in Gauss/Seidel chaotic relaxation. But what if it's not OK? How can we tell? What methods or tools can be used to help us diagnose the problem and find a solution?

Let's take a look...

Wrong answer!

We've now invested all this time and energy in parallelizing our program using message passing, so why can't we 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 our program give correct answers based on the problem description, but that doesn't mean they both can't compute different answers! Let's examine the problem more closely by running the **chaotic.f** program under the **pedb** debugger:

```
$ pedb chaotic
```

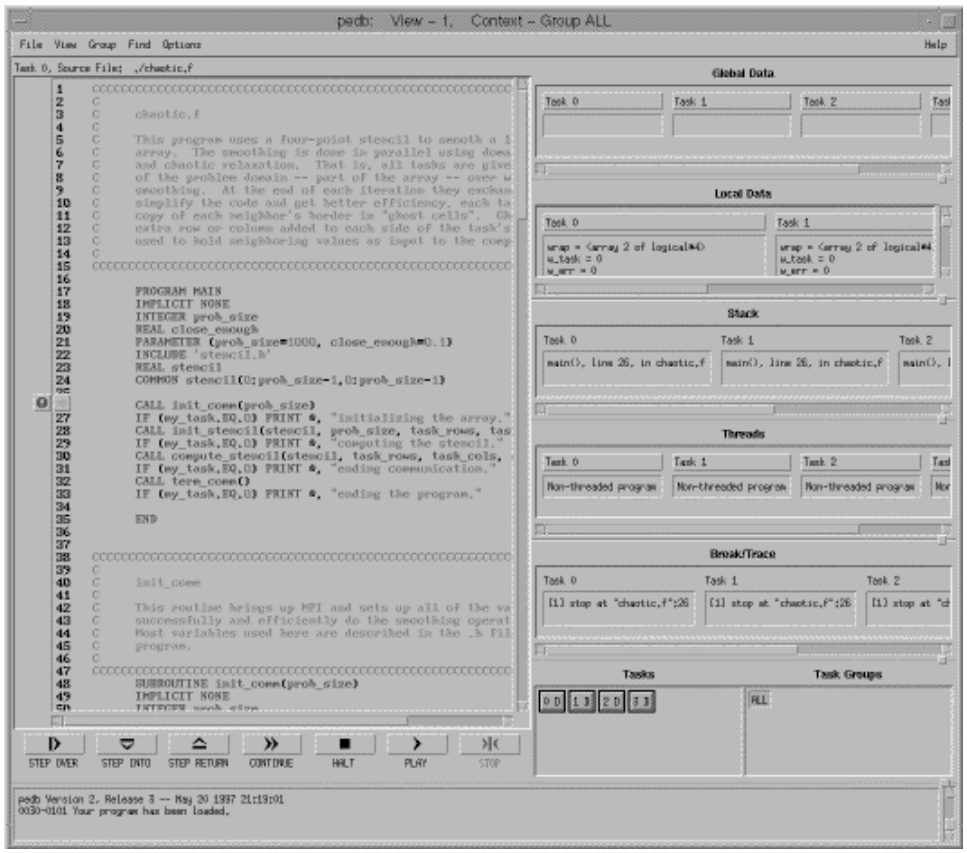

Figure 13. pedb main window

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

Let's run each version under the control of the debugger to view and compare the results of our arrays.

With this test, we 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's the serial (reordered) array and parallel (chaotic) array stencils:

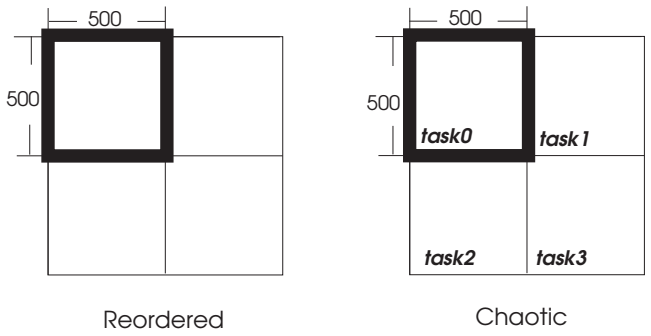

Figure 14. Serial and Parallel Array Stencils

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

PLACE the mouse cursor on the line 168.

DOUBLE CLICK the left mouse button.

▲ The debugger sets a breakpoint at line 168. A stop marker (drawn to look like a stop sign containing an exclamation point) appears next to the line.

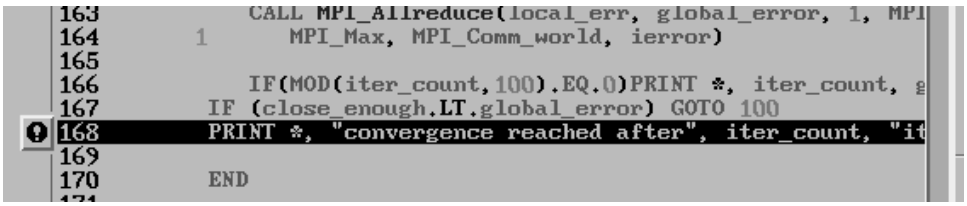

Figure 15. Setting Breakpoint

Note: 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.

Next, we'll need to examine the array stencil. We can do this by exporting the array to a file to compare answers.

First, select the variable *stencil* on task 0.

PLACE the mouse cursor over the *stencil* variable.

CLICK the left mouse button to highlight the selection.

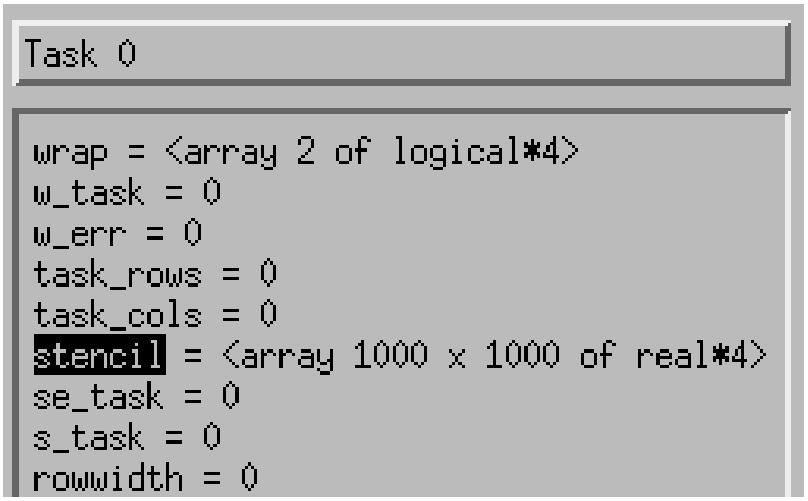

Figure 16. Variable List

Bring up the Variable Options menu.

CLICK the right mouse button.

▲ The Variable Options menu appears.

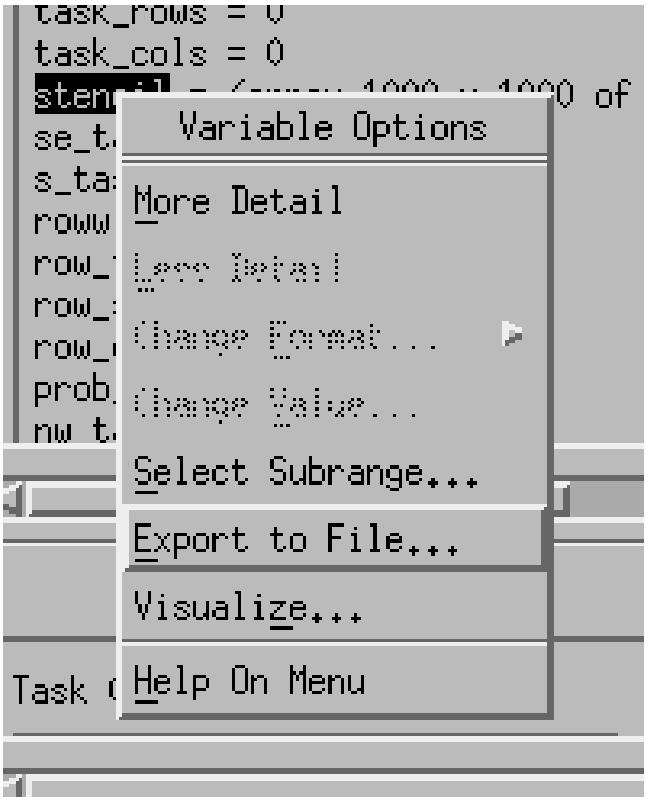

Figure 17. Variable Options Menu

SELECT *Export to File*

▲ The Export window appears.

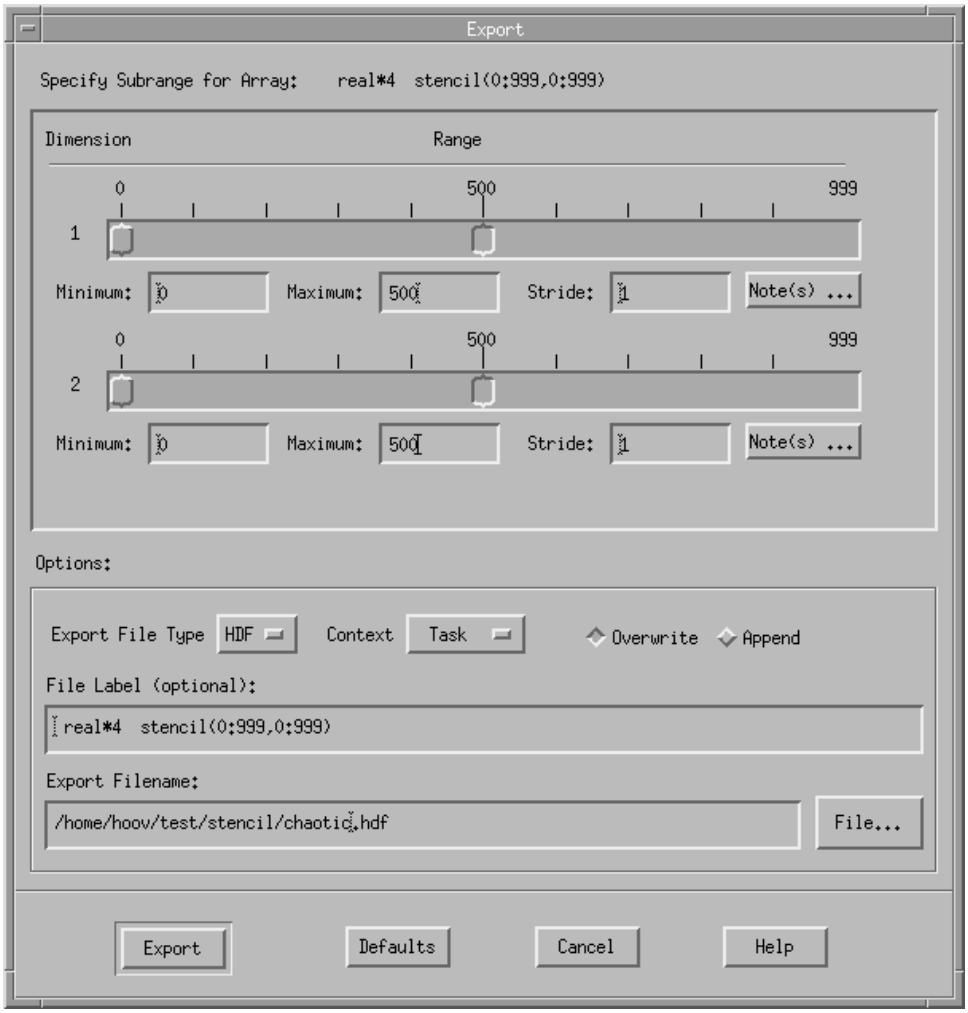

Figure 18. Export Window

We'll be exporting the subrange 0 through 500 in both dimensions.

Next, rename the file to be created to **chaotic.hdf** so that we'll be able to differentiate our results from the two runs. Then, press the *Export* button to export the array to an HDF file. Wait for the export to complete and then exit the debugger.

As you can see, we now have a file called **chaotic.hdf** in our current working directory.

Remember that our objective is to compare each of the results, so we should set our number of processes to 1 and then run the **reordered** program in the same way as we did with **chaotic**.

Note: Remember to only export array indices 0 through 500 in each dimension to get a valid comparison. Also, remember to rename the output file to **reorder.hdf**.

And now for something completely different...let's take a look at our HDF files using IBM Visualization Data Explorer (this is a separate program product, available from IBM). Here's what we see:

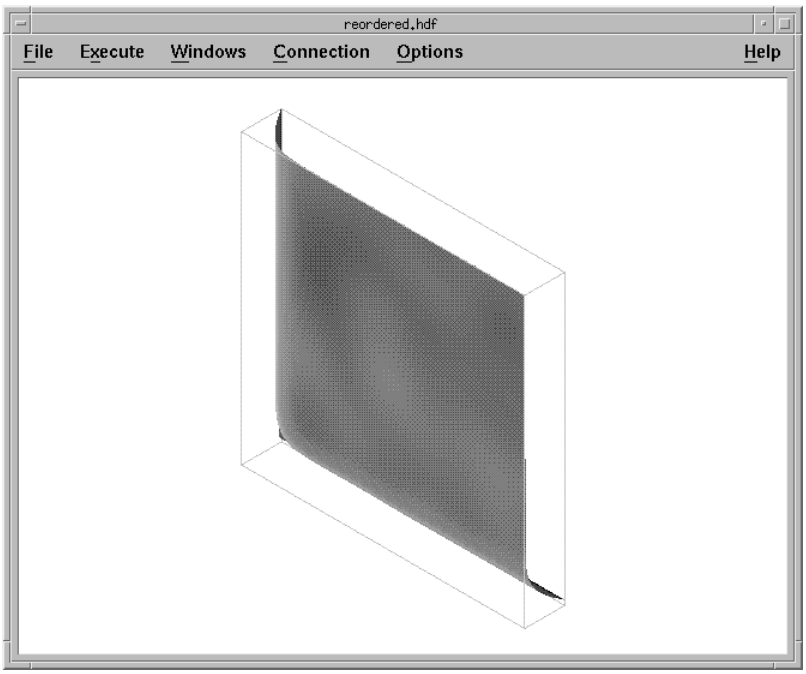

Figure 19. Visualizing Reordered HDF File

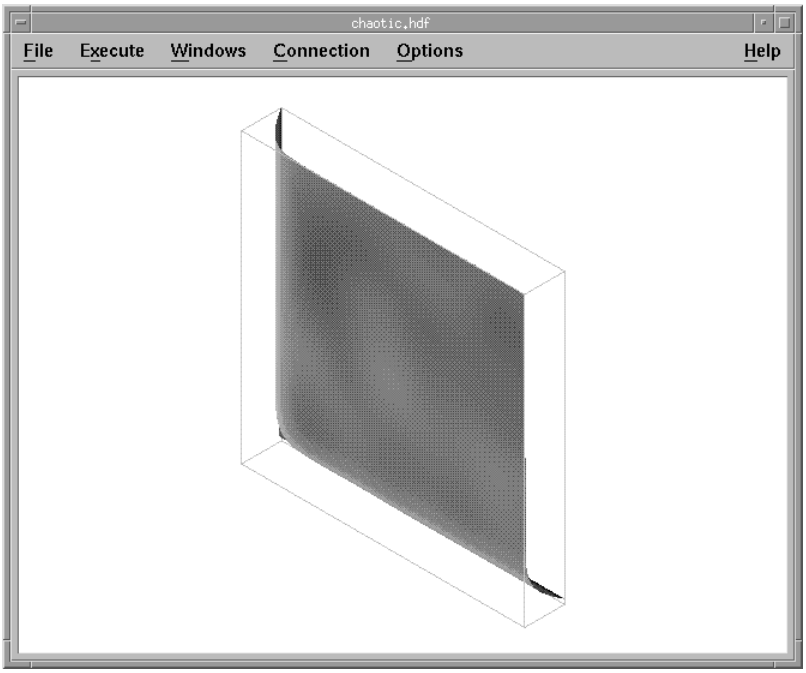

Figure 20. Visualizing Chaotic HDF File

Well, they look the same, but how do we really know? To find out, we can look at the **reordered** and **chaotic** raw data. They appear to be the same, but if we look more closely, we can see some differences. Let's take a look at a section of row 2 in the raw data of each.

Here's the **reordered** data:

```
(row, col)
 (2,310)  (2,311)  (2,312)  (2,313)  (2,314)  (2,315)
9.481421 9.440039 9.398028 9.355416 9.312231 9.268500

 (2,316)  (2,317)  (2,318)  (2,319)  (2,320)  (2,321)
9.224252 9.179513 9.134314 9.088681 9.042644 8.996230
```

Here's the **chaotic** data:

```
(row, col)
 (2,310)  (2,311)  (2,312)  (2,313)  (2,314)  (2,315)
9.481421 9.440039 9.398028 9.355416 9.312232 9.268501

 (2,316)  (2,317)  (2,318)  (2,319)  (2,320)  (2,321)
9.224253 9.179514 9.134315 9.088682 9.042645 8.996231
```

After looking at the raw data, we see that our answers are definitely similar, but different. Why? We can blame it on a couple of things, but it's mostly due to the chaotic nature of our algorithm. If we look at how the average is being computed in the serial version of our program, we can see that within each iteration of our loop, two array cells are from the old iteration and two are from new ones.

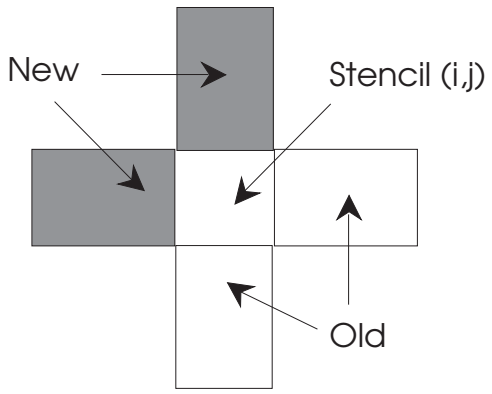

Figure 21. How the average is computed in a 4-point stencil

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 our program, this is true for the interior array cells but not for our shared boundaries. For more information, you may find *In Search of Clusters* by Gregory F. Phister, helpful. See “Related Non-IBM Publications” on page xiii.

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

Here's the Fix!

So, you've got a serial and parallel program that don't give you the same answers. One way to fix this is to skew the processing of the global array. We 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.

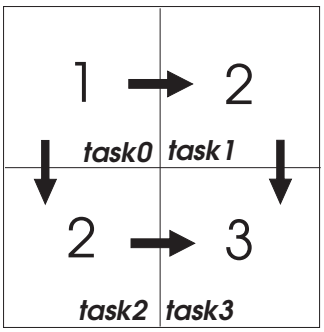

Figure 22. Sequence of Array Calculation

The only thing we need to modify in our 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 our serial version.

Here's our modified (skewed) parallel program. It's 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
```

Program Name	Tasks	Wallclock Time	Array Size per Task
naive	1 (single processor)	11m1.94s	1000x1000
reordered	1 (single processor)	5m35.38s	1000x1000
chaotic	4 (processors)	2m4.58s	500x500
skewed	4 (processors)	4m41.87s	500x500

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

It's Still Not Fast Enough!

We've got the right answers now, but we still want our program to move faster, so we're not done yet. Let's look at our new code to see what other techniques we can use to speed up execution. We'll look at:

- Convergence rates (total number of iterations)
- Load balance
- Synchronization/communication time

Let's use VT to see how our program executes:

```
$ skewed -tracelevel 3
```

Note: We use trace level 3 to look at message passing traits.

```
$ vt  skewed.trc
```

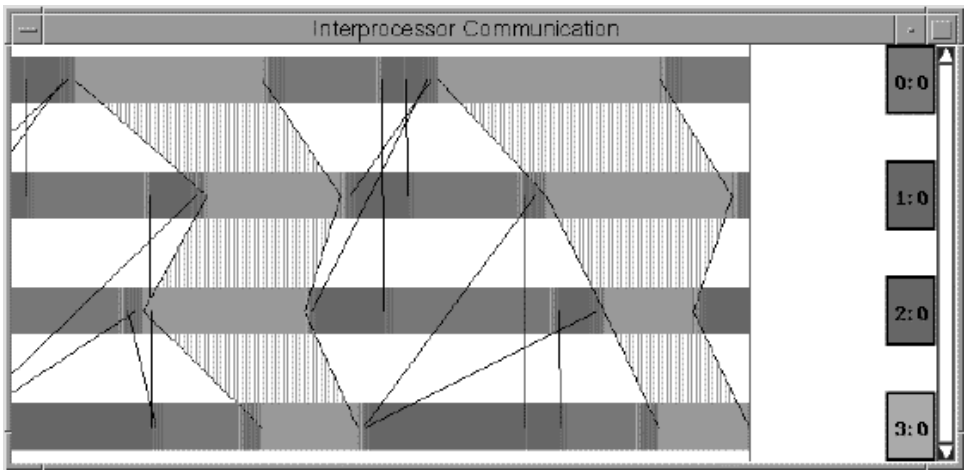

Figure 23. VT Space Time Diagram

By looking at the message passing, we can see some peculiar characteristics of our program. For instance, 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 we've 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 in order 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, we'll 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 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.

Now that we've made these improvements to our program, we'll most likely see improvement in our load balance as well.

Let's see how we affected our load balance by running our new code with tracing turned on.

```
$ pipelined -tracelevel 3
```

Our new code shows the following message passing graph:

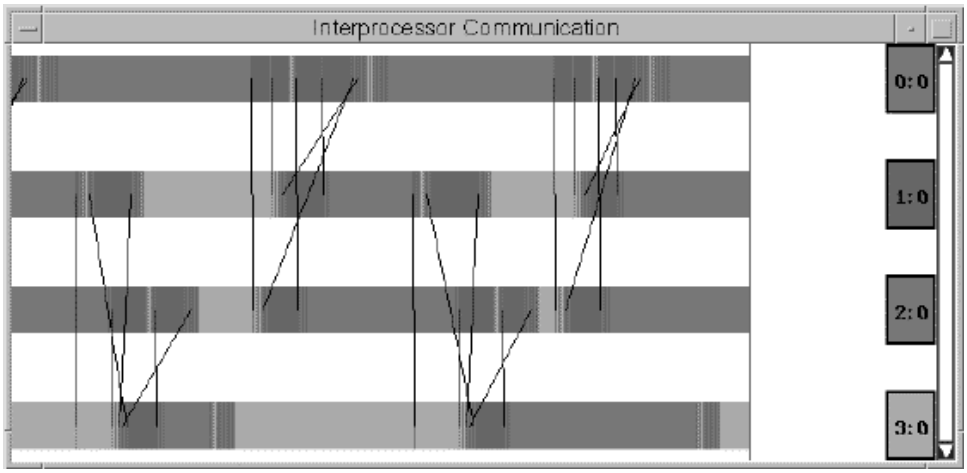

Figure 24. VT Displays

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

```
$ time poe pipelined
```

Program Name	Tasks	Wallclock Time	Array Size per Task
naive	1 (single processor)	11m1.94s	1000x1000
reordered	1 (single processor)	5m35.38ss	1000x1000
chaotic	4 (processors)	2m4.58s	500x500
skewed	4 (processors)	4m41.87s	500x500
pipelined	4 (processors)	2m7.42s	500x500

So, how did we do? Here are the final statistics:

- From the naive serial program to the reordered serial program there was a speedup of 197%
- From the reordered serial program to the pipelined parallel program there was a speedup of 263%
- From the naive serial program to the pipelined parallel program there was a speedup of 519%

As you can see, we were able to significantly improve the performance of our program and, at the same time, get a consistent, correct answer. And all we had when we started was our serial program!

Tuning Summary

As you've probably deduced, 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 the IBM Parallel Environment for AIX has taken toward performance tuning is to provide tools which give you the information necessary to perform the tuning. We've given you the fish. Where you go from there is up to you.

Chapter 5. Babel fish

This chapter provides information that will help you translate your MPL parallel program into a program that conforms to the MPI standard. In particular, it tells you which MPI calls to substitute for the ones you use right now in MPL.

In *The Hitchhiker's Guide to the Galaxy*, the *Babel Fish* is a tiny fish that, when inserted into your ear, can make any language understandable to you. Well, it's not quite that easy to translate a parallel program, but this chapter at least helps you determine how to perform the equivalent or comparable function in MPI that you did with MPL.

Note that the syntax in this section is in C unless we tell you otherwise. For the corresponding FORTRAN MPI syntax, see *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*. For the corresponding FORTRAN MPL syntax, see *IBM Parallel Environment for AIX: MPL Programming and Subroutine Reference*. Another document that may be helpful is *A Message-Passing Interface Standard, Version 1.1*, available from the University of Tennesee.

Point-to-Point Communication

SEND (Non-Blocking)

MPL/MPI	Description
MPL	mpc_send(&buf,msglen,dest,tag,&msgid)
MPI	MPI_Isend(&buf,count,datatype,dest,tag,comm,&request)

RECEIVE (Non-Blocking)

MPL/MPI	Description
MPL	mpc_recv(&buf,msglen,&source,&tag,&msgid)
MPI	MPI_Irecv(&buf,count,datatype,source,tag,comm,&request)

SEND (Blocking)

MPL/MPI	Description
MPL	mpc_bsend(&buf,msglen,dest,tag)
MPI	MPI_Send(&buf,count,datatype,dest,tag,comm)
Note: Don't confuse MPI_Bsend with MPI_Send. MPI_Bsend is a BUFFERED send, not a BLOCKING send.	

© Copyright IBM Corp. 1997

RECEIVE (Blocking)

MPL/MPI	Description
MPL	mpc_brecv(&buf,msglen,&source,&tag,&nbytes)
MPI	MPI_Recv(&buf,count,datatype,source,tag,comm,&status)

SEND/RECEIVE (Blocking)

MPI/MPL	Description
MPL	mpc_bsendrecv(&sendbuf,sendlen,dest,tag,&recvbuf,recvlen,&source,&nbytes)
MPI	MPI_Sendrecv(&sendbuf,sendcount,sendtype,dest,tag,&recvbuf,recvcount,recvtype, source,tag,comm,&status)

STATUS

MPI/MPL	Description
MPL	nbytes = mpc_status(msgid)
MPI	MPI_Get_count(&status,MPI_BYTE,&nbytes)

WAIT

MPI/MPL	Description
MPL	mpc_wait(&msgid,&nbytes)
MPI	For a specific msgid: • MPI_Wait(&request,&status) For msgid = DONTCARE: • MPI_Waitany(count,requests,&index,&status) • The requests array must be maintained by the user. For msgid = ALLMSG: • MPI_Waitall(count,requests,statuses) • The requests array must be maintained by the user.

TASK_SET

MPI/MPL	Description
MPL	mpc_task_set(nbuf,stype)
MPI	Truncation Mode: • No MPI equivalent. Can be simulated by setting the error handler to "return": MPI_Errhandler_set(comm,MPI_ERRORS_RETURN); and testing the return code for receives, waits for receives, etc.: MPI_Error_class(rc,&class); if(class != MPI_ERR_TRUNCATE) { (handle error) } Develop/Run Mode: • Enable DEVELOP mode by setting MP_EUIDEVELOP environment variable to YES. Buffer Mode: • Use MPI_Buffer_attach.

TASK_QUERY

MPI/MPL	Description
MPL	mpc_task_query(nbuf,nelem,qtype)
MPI	Truncation Mode: • No MPI equivalent Message Type Bounds: lower bound = 0 upper bound: int *valptr; MPI_Attr_get(MPI_COMM_WORLD,MPI_TAG_UB,&valptr,&flag) tag_up_bound = *valptr; Wildcards: **ALLGRP (0)** MPI_COMM_WORLD **DONTCARE (-1)** MPI_ANY_SOURCE, MPI_ANY_TAG **ALLMSG (-2)** No MPI equivalent - see mpc_wait **NULLTASK (-3)** MPI_PROC_NULL

ENVIRON

MPI/MPL	Description
MPL	mpc_environ(&numtask,&taskid)
MPI	MPI_Comm_size(MPI_COMM_WORLD,&numtask) MPI_Comm_rank(MPI_COMM_WORLD,&taskid)

STOPALL

MPI/MPL	Description
MPL	mpc_stopall(errcode)
MPI	MPI_Abort(comm,errcode)

PACK

MPI/MPL	Description
MPL	mpc_pack(&inbuf,&outbuf,blklen,offset,blknum)

MPI	MPI_Type_hvector(1,blklen,offset,MPI_BYTE,&datatype)
	position = 0;
	outcount = (blknum-1)*offset + blklen;
	MPI_Pack(&inbuf,blknum,datatype,&outbuf,outcount,&position,comm)

UNPACK

MPI/MPL	Description
MPL	mpc_unpack(&inbuf,&outbuf,blklen,offset,blknum)

MPI	MPI_Type_hvector(1,blklen,offset,MPI_BYTE,&datatype)
	position = 0;
	insize = (blknum-1)*offset + blklen;
	MPI_Unpack(&inbuf,insize,&position,&outbuf,blknum,datatype,comm)

VSEND (Blocking)

MPI/MPL	Description
MPL	mpc_bvsend(&buf,blklen,offset,blknum,dest,tag)
MPI	MPI_Type_hvector(1,blklen,offset,MPI_BYTE,&datatype) MPI_Send(&buf,blknum,datatype,dest,tag,comm)

VRECV (Blocking)

MPI/MPL	Description
MPL	mpc_bvrecv(&buf,blklen,offset,blknum,&source,&tag,&nbytes)
MPI	MPI_Type_hvector(1,blklen,offset,MPI_BYTE,&datatype) MPI_Recv(&buf,blknum,datatype,source,tag,comm,&status)

PROBE

MPI/MPL	Description
MPL	mpc_probe(&source,&tag,&nbytes)
MPI	MPI_Iprobe(source,tag,comm,&flag,&status)
Note: MPI also provides a blocking version of probe: **MPI_Probe**, which can be substituted for an MPL probe in an infinite loop.	

Collective Communications

BROADCAST

MPI/MPL	Description
MPL	mpc_bcast(&buf,msglen,root,gid)
MPI	MPI_Bcast(&buf,count,datatype,root,comm)

COMBINE

MPI/MPL	Description
MPL	mpc_combine(&sendbuf,&recvbuf,msglen,func,gid)
MPI	MPI_Allreduce(&sendbuf,&recvbuf,count,datatype,op,comm)
Note: See “Reduction Functions” on page 98.	

CONCAT

MPI/MPL	Description
MPL	mpc_concat(&sendbuf,&recvbuf,blklen,gid)
MPI	MPI_Allgather(&sendbuf,sendcount,sendtype,&recvbuf,recvcount,recvtype,comm)

GATHER

MPI/MPL	Description
MPL	mpc_gather(&sendbuf,&recvbuf,blklen,root,gid)
MPI	MPI_Gather(&sendbuf,count,datatype,&recvbuf,count,datatype,root,comm)

INDEX

MPI/MPL	Description
MPL	mpc_index(&sendbuf,&recvbuf,blklen,gid)
MPI	MPI_Alltoall(&sendbuf,count,datatype,&recvbuf,count,datatype,comm)

PREFIX

MPI/MPL	Description
MPL	mpc_prefix(&sendbuf,&recvbuf,msglen,func,gid)
MPI	MPI_Scan(&sendbuf,&recvbuf,count,datatype,op,comm)
Note: See “Reduction Functions” on page 98.	

REDUCE

MPI/MPL	Description
MPL	mpc_reduce(&sendbuf,&recvbuf,msglen,root,func,gid)
MPI	MPI_Reduce(&sendbuf,&recvbuf,count,datatype,op,root,comm)
Note: See “Reduction Functions” on page 98.	

SCATTER

MPI/MPL	Description
MPL	mpc_scatter(&sendbuf,&recvbuf,blklen,root,gid)
MPI	MPI_Scatter(&sendbuf,count,datatype,&recvbuf,count,datatype,root,comm)

SHIFT

MPI/MPL	Description
MPL	mpc_shift(&sendbuf,&recvbuf,msglen,step,flag,gid)
MPI	MPI_Cart_shift(comm,direction,step,&source,&dest) MPI_Sendrecv(&sendbuf,count,datatype,dest,tag,&recvbuf,count,datatype, source,tag,comm,&status);
Note: *comm* must be a communicator with a cartesian topology. See **MPI_CART_CREATE** in *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*.	

SYNC

MPI/MPL	Description
MPL	mpc_sync(gid)
MPI	MPI_Barrier(comm)

GETLABEL

MPI/MPL	Description
MPL	mpc_getlabel(&label,gid)
MPI	No MPI equivalent. Can be simulated by creating a label attribute key with **MPI_Keyval_create**, attaching a label attribute to a communicator with **MPI_Attr_put**, and retrieving it with **MPI_Attr_get**.

GETMEMBERS

MPI/MPL	Description
MPL	mpc_getmembers(&glist,gid)
MPI	MPI_Comm_group(MPI_COMM_WORLD,&group_world) MPI_Group_size(group_world,&gsize) for(i=0;i<gsize;i++) ranks]i[= i; MPI_Group_translate_ranks(group,gsize,&ranks,group_world,&glist)

GETRANK

MPI/MPL	Description
MPL	mpc_getrank(&rank,taskid,gid)
MPI	MPI_Comm_group(MPI_COMM_WORLD,&group_world) MPI_Group_translate_ranks(group_world,1,&taskid,group2,&rank)

GETSIZE

MPI/MPL	Description
MPL	mpc_getsize(&gsize,gid)
MPI	MPI_Group_size(group,&gsize)

GETTASKID

MPI/MPL	Description
MPL	mpc_gettaskid(rank,&taskid,gid)
MPI	MPI_Comm_group(MPI_COMM_WORLD,&group_world) MPI_Group_translate_ranks(group1,1,&rank,group_world,&taskid)

GROUP

MPI/MPL	Description
MPL	mpc_group(gsize,&glist,label,&gid)
MPI	MPI_Comm_group(MPI_COMM_WORLD,&group_world) MPI_Group_incl(group_world,gsize,&glist,&gid)

PARTITION

MPI/MPL	Description
MPL	mpc_partition(parent_gid,key,label,&gid)
MPI	MPI_Comm_split(comm,label,key,&newcomm)

Reduction Functions

MPL Function	MPI Equivalent
i_vadd	**Operator**: MPI_SUM **Datatype**: MPI_INT, MPI_INTEGER
s_vadd	**Operator**: MPI_SUM **Datatype**: MPI_FLOAT, MPI_REAL
d_vadd	**Operator**: MPI_SUM **Datatype**: MPI_DOUBLE, MPI_DOUBLE_PRECISION
i_vmul	**Operator**: MPI_PROD **Datatype**: MPI_INT, MPI_INTEGER
s_vmul	**Operator**: MPI_PROD **Datatype**: MPI_FLOAT, MPI_REAL
d_vmul	**Operator**: MPI_PROD **Datatype**: MPI_DOUBLE, MPI_DOUBLE_PRECISION
i_vmax	**Operator**: MPI_MAX **Datatype**: MPI_INT, MPI_INTEGER
s_vmax	**Operator**: MPI_MAX **Datatype**: MPI_FLOAT, MPI_REAL
d_vmax	**Operator**: MPI_MAX **Datatype**: MPI_DOUBLE, MPI_DOUBLE_PRECISION

MPL Function	MPI Equivalent
i_vmin	**Operator**: MPI_MIN **Datatype**: MPI_INT, MPI_INTEGER
s_vmin	**Operator**: MPI_MIN **Datatype**: MPI_FLOAT, MPI_REAL
d_vmin	**Operator**: MPI_MIN **Datatype**: MPI_DOUBLE, MPI_DOUBLE_PRECISION
b_vand	**Operator**: MPI_BAND **Datatype**: MPI_BYTE
b_vor	**Operator**: MPI_BOR **Datatype**: MPI_BYTE
b_vxor	**Operator**: MPI_BXOR **Datatype**: MPI_BYTE
l_vand	**Operator**: MPI_LAND **Datatype**: MPI_BYTE
l_vor	**Operator**: MPI_LOR **Datatype**: MPI_BYTE
Note: The **count** parameter can be computed as follows: MPI_Type_size(datatype,&size) count = msglen/size;	

User-Defined Reduction Functions

MPL/MPI	Description
MPL	void func(&inbuf1,&inbuf2,&outbuf,&len) Note that *func* is passed as an argument to the Collective Communication Library (CCL) function.
MPI	void func(&inbuf,&inoutbuf,&count,&datatype) MPI_Op_create(func,commute,&op) Note that *op* is passed as an argument to the CCL function.

Global Variables and Constants

Last Error Code

MPL/MPI	Description
MPL	mperrno
MPI	No equivalent; error codes are returned by each function.

Wildcards

MPL Wildcard	MPI Equivalent
ALLGRP (0)	MPI_COMM_WORLD
DONTCARE (-1)	MPI_ANY_SOURCE, MPI_ANY_TAG
ALLMSG (-2)	no MPI equivalent - see mpc_wait
NULLTASK (-3)	MPI_PROC_NULL

General Notes

This section provides some specific things to keep in mind when translating your program from MPL to MPI.

Task Indentifiers

In MPL, task identifiers such as *src* and *dest* are absolute task ids. In MPI, they are ranks within a communicator group. For the communicator **MPI_COMM_WORLD**, they are the same.

Message Length

- In MPL, message lengths are expressed in bytes. In MPI, they are expressed as *count,datatype*. Thus, a message consisting of ten 4-byte integers would be coded as *40* in MPL, and as *10,MPI_INT* or *10,MPI_INTEGER* in MPI.
- For send and receive operations, MPL returned the message length in the *nbytes* parameter. MPI returns this information in *status*. It can be accessed as follows:

```
MPI_Get_count(&status,MPI_BYTE,&nbytes)
```

Creating MPI Objects

MPI Objects should be created as follows:

Object	C	FORTRAN
Communicators	MPI_Comm commid	integer commid
Groups	MPI_Group groupid	integer groupid
Requests	MPI_Request requestid	integer reqestid
Reduction Ops	MPI_Op opid	integer opid
Error Handlers	MPI_Errhandler handlerid	integer handlerid
Data Types	MPI_Datatype typeid	integer typeid
Attribute Keys	int keyid	integer keyid
Status	MPI_Status status	integer status(MPI_STATUS_SIZE)

Using Wildcard Receives

For wildcard receives, MPL backstuffed the actual source and message type into the addresses of these parameters supplied with the receive call. In MPI, the actual values are returned in the *status* parameter, and may be retrieved as follows.

For programs written in C:

```
source = status.MPI_SOURCE;
tag    = status.MPI_TAG;
```

For programs written in FORTRAN:

```
source = status(MPI_SOURCE)
tag    = status(MPI_TAG)
```

Also note the following for C applications. In MPL, the *source* and *type* parameters were passed by reference, whereas in MPI, they are passed by value.

Reduction Functions

In MPI, user-defined reduction functions can be defined as commutative or non-commutative (see **MPI_Op_create**), whereas in MPL, all reduction functions were assumed to be commutative. Reduction functions must be associative in both MPL and MPI.

Error Handling

In MPL, C functions provided return codes that could be checked to determine if an error occurred, and FORTRAN functions printed error messages and terminated the job. In MPI, the default for both C and FORTRAN is to print a message and terminate the job. If return codes are desired, the error handler must be set as follows (per communicator):

```
MPI_Errhandler_set(comm,MPI_ERRORS_RETURN);
```

In FORTRAN, error codes are returned in the last parameter of each function, *ierror*.

Also, IBM's MPI implementation provides a third predefined error handler, **MPE_ERRORS_WARN**, which prints a message and returns an error code without terminating the job. In DEVELOP mode, messages are always printed.

Mixing MPL and MPI Functions in the Same Application

MPL and MPI functions can be used in the same application (using the non-threaded library only), but the following rules must be followed:

- Messages sent by MPL must be received by MPL, and messages sent by MPI must be received by MPI.
- For any given invocation of a CCL call, all applicable tasks must use the same API: either MPL or MPI. It is illegal for some tasks to use MPL, and others to use MPI.
- Objects are only meaningful to the API which generated them. For example, a request ID that is returned by **MPI_Isend** cannot be used with **mpc_wait**. Also, **ALLGRP** should not be used as a communicator, and **MPI_COMM_WORLD** should not be used as an MPL group ID. Care should be taken not to carry concepts from one API to the other.

The same DEVELOP MODE environment variable, MPI_EUIDEVELOP is used by MPL and MPI. If it is set to YES, then DEVELOP MODE is turned on for **both** MPL and MPI.

Before and After Using MPI Functions

All application programs that use MPI functions **must** call **MPI_Init** before calling any other MPI function (except **MPI_Initialized**). All applications that use MPI functions **should** call **MPI_Finalize** as the last MPI call they make. Failure to do this may make the application non-portable.

If an application makes no MPI calls, then it is not necessary for it to call **MPI_Init** or **MPI_Finalize**.

Using Message Passing Handlers

Only a subset of MPL message passing is allowed on handlers that are created by the MPL Receive and Call function (**mpc_rcvncall** or **MP_RCVNCALL**). MPI calls on these handlers are not supported.

Appendix A. A Sample Program to Illustrate Messages

This appendix provides sample output for a program run under POE with the maximum level of message reporting. It also points out the different types of messages you can expect, and explains what they mean.

To set the level of messages that get reported when you run your program, you can use the **-infolevel** (or **-ilevel**) option when you invoke POE, or 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 for AIX: Operation and Use, Vol. 1*.

Note that we're using numbered prefixes along the left-hand edge of the output you see below as a way to refer to particular lines; they are **not** part of the output you'll 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 105.

The following command:

```
> poe hello_world_c -procs 2 -hostfile pool.list -infolevel 6
```

produces the following output:

```
1   INFO: DEBUG_LEVEL changed from 0 to 4
2   D1<L4>: Open of file pool.list successful
3   D1<L4>: mp_euilib = ip
4   D1<L4>: task 0 5 1
5   D1<L4>: extended 1  5 1
6   D1<L4>: node allocation strategy = 2
7   INFO: 0031-690  Connected to Resource Manager
8   INFO: 0031-118  Pool 1 requested for task 0
9   INFO: 0031-118  Pool 1 requested for task 1
10  D1<L4>: Elapsed time for call to jm_allocate: 0 seconds
11  INFO: 0031-119  Host k10n01.ppd.pok.ibm.com allocated for task 0
12  INFO: 0031-119  Host k10n02.ppd.pok.ibm.com allocated for task 1
13  D1<L4>: Requesting service pmv2
14  D1<L4>: Jobid = 803755221
15  D4<L4>: Command args:<>
16  D1<L4>: Task 0 pulse count is 0
17  D1<L4>: Task 1 pulse count is 0
18  D3<L4>: Message type 34 from source 0
19  D1<L4>: Task 0 pulse count is 1
20  D1<L4>: Task 1 pulse count is 0
21  D3<L4>: Message type 21 from source 0
22    0:INFO: 0031-724  Executing program: <hello_world_c>
23  D3<L4>: Message type 34 from source 1
24  D1<L4>: Task 0 pulse count is 1
25  D1<L4>: Task 1 pulse count is 1
26  D3<L4>: Message type 21 from source 0
27    0:INFO: DEBUG_LEVEL changed from 0 to 4
28    0:D1<L4>: mp_euilib is <ip>
29    0:D1<L4>: mp_css_interrupt is <0>
30  D3<L4>: Message type 21 from source 1
31    1:INFO: 0031-724  Executing program: <hello_world_c>
32  D3<L4>: Message type 21 from source 0
33    0:D1<L4>: cssAdapterType is <1>
34  D3<L4>: Message type 21 from source 1
35    1:INFO: DEBUG_LEVEL changed from 0 to 4
36    1:D1<L4>: mp_euilib is <ip>
```

© Copyright IBM Corp. 1997

```
37    1:D1<L4>: mp_css_interrupt is <0>
38    1:D1<L4>: cssAdapterType is <1>
39  D3<L4>: Message type 23 from source 0
40  D1<L4>: init_data for task 0: <129.40.161.65:1675>
41  D3<L4>: Message type 23 from source 1
42  D1<L4>: init_data for task 1: <129.40.161.66:1565>
43  D2<L4>: About to call pm_address
44  D2<L4>: Elapsed time for pm_address: 0 seconds
45  D3<L4>: Message type 21 from source 1
46    1:D1<L4>: About to call mpci_connect
47  D3<L4>: Message type 21 from source 0
48  D3<L4>: Message type 21 from source 1
49  D3<L4>: Message type 21 from source 0
50  D3<L4>: Message type 21 from source 1
51  D3<L4>: Message type 21 from source 0
52  D3<L4>: Message type 21 from source 1
53  D3<L4>: Message type 21 from source 0
54  D3<L4>: Message type 21 from source 1
55  D3<L4>: Message type 21 from source 0
56  D3<L4>: Message type 21 from source 1
57  D3<L4>: Message type 21 from source 0
58  D3<L4>: Message type 21 from source 1
59  D3<L4>: Message type 21 from source 0
60  0:D1<L4>: About to call mpci_connect
61  D3<L4>: Message type 21 from source 1
62  D3<L4>: Message type 21 from source 0
63  D3<L4>: Message type 21 from source 1
64  D3<L4>: Message type 21 from source 0
65  D3<L4>: Message type 21 from source 1
66    1:D1<L4>: Elapsed time for mpci_connect: 1 seconds
67  D3<L4>: Message type 21 from source 0
68  D3<L4>: Message type 44 from source 1
69  D3<L4>: Message type 21 from source 0
70  D3<L4>: Message type 21 from source 0
71    0:D1<L4>: Elapsed time for mpci_connect: 0 seconds
72  D3<L4>: Message type 44 from source 0
73  D2<L4>: <C O N N E C T   D A T A>
74  D2<L4>: Task      Down Count Nodes
75  D2<L4>: ====      ========== =====
76  D2<L4>: 0    0
77  D2<L4>: 1    0
78  D2<L4>: <E N D   O F   C O N N E C T   D A T A>
79  D3<L4>: Message type 21 from source 0
80  0:D1<L4>: About to call _ccl_init
81  D3<L4>: Message type 21 from source 1
82  D3<L4>: Message type 21 from source 1
83  D3<L4>: Message type 21 from source 1
84  D3<L4>: Message type 21 from source 1
85  D3<L4>: Message type 21 from source 1
86  D3<L4>: Message type 21 from source 1
87  D3<L4>: Message type 21 from source 1
88    1:D1<L4>: About to call _ccl_init
89  D3<L4>: Message type 21 from source 0
90  D3<L4>: Message type 21 from source 1
91  D3<L4>: Message type 21 from source 0
92  D3<L4>: Message type 21 from source 0
93  D3<L4>: Message type 21 from source 1
94  D3<L4>: Message type 21 from source 0
95  D3<L4>: Message type 21 from source 1
96  D3<L4>: Message type 21 from source 0
97  D3<L4>: Message type 21 from source 1
98  D3<L4>: Message type 21 from source 0
99  D3<L4>: Message type 21 from source 1
100 D3<L4>: Message type 21 from source 0
```

```
101 D3<L4>: Message type 21 from source 1
102 D3<L4>: Message type 21 from source 0
103 D3<L4>: Message type 21 from source 1
104 D3<L4>: Message type 21 from source 0
105   0:D1<L4>: Elapsed time for _ccl_init: 0 seconds
106 D3<L4>: Message type 21 from source 1
107 D3<L4>: Message type 20 from source 0
108   0:Hello, World!
109 D3<L4>: Message type 21 from source 1
110   1:D1<L4>: Elapsed time for _ccl_init: 0 seconds
111 D3<L4>: Message type 21 from source 0
112  0:INFO: 0033-3075 VT Node Tracing completed.  Node merge beginning
113 D3<L4>: Message type 20 from source 1
114   1:Hello, World!
115 D3<L4>: Message type 21 from source 0
116   0:INFO: 0031-306  pm_atexit: pm_exit_value is 0.
117 D3<L4>: Message type 21 from source 1
118   1:INFO: 0033-3075 VT Node Tracing completed.  Node merge beginning
119 D3<L4>: Message type 17 from source 0
120 D3<L4>: Message type 21 from source 1
121   1:INFO: 0031-306  pm_atexit: pm_exit_value is 0.
122 D3<L4>: Message type 17 from source 1
123 D3<L4>: Message type 22 from source 0
124 INFO: 0031-656  I/O file STDOUT closed by task 0
125 D3<L4>: Message type 22 from source 1
126 INFO: 0031-656  I/O file STDOUT closed by task 1
127 D3<L4>: Message type 15 from source 0
128 D1<L4>: Accounting data from task 0 for source 0:
129 D3<L4>: Message type 15 from source 1
130 D1<L4>: Accounting data from task 1 for source 1:
131 D3<L4>: Message type 22 from source 0
132 INFO: 0031-656  I/O file STDERR closed by task 0
133 D3<L4>: Message type 22 from source 1
134 INFO: 0031-656  I/O file STDERR closed by task 1
135 D3<L4>: Message type 1 from source 0
136 INFO: 0031-251  task 0 exited: rc=0
137 D3<L4>: Message type 1 from source 1
138 INFO: 0031-251  task 1 exited: rc=0
139 D1<L4>: All remote tasks have exited: maxx_errcode = 0
140 INFO: 0031-639  Exit status from pm_respond = 0
141 D1<L4>: Maximum return code from user = 0
142 D2<L4>: In pm_exit... About to call pm_remote_shutdown
143 D2<L4>: Sending PMD_EXIT to task 0
144 D2<L4>: Sending PMD_EXIT to task 1
145 D2<L4>: Elapsed time for pm_remote_shutdown: 0 seconds
146 D2<L4>: In pm_exit... About to call jm_disconnect
147 D2<L4>: Elapsed time for jm_disconnect: 0 seconds
148 D2<L4>: In pm_exit... Calling exit with status = 0 at Wed Jun 21 07:15:07 19
```

Figuring Out What All of This Means

When you set **-infolevel** to 6, you get the full complement of diagnostic messages, which we'll explain here.

The example above includes numbered prefixes along the left-hand edge of the output so that we can refer to particular lines, and then tell you what they mean. Remember, these prefixes are **not** part of your output. The table below points you to the line number of the messages that are of most interest, and provides a short explanation.

Line(s)	Message Description
7, 8, 9	Pool 1 was requested in host.list file, *pool.list*.
11-12	Names hosts that are used.
13	Indicates that service pmv2, from /**etc**/**services** is being used.
18	Message type 34 indicates *pulse* activity (the pulse mechanism checked that each remote node was actively participating with the home node).
21	Message type 21 indicates a STDERR message.
28	Indicates that the **euilib** message passing protocol was specified.
40, 42	String returned from **_eui_init**, which initializes **mpci_libarary**.
66, 71	Indicates initialization of **mpci_library**.
107, 108, 113, 114	Message type 20 shows STDOUT from your program.
116, 121	Indicates that the user's program has reached the exit handler. The exit code is 14.
119, 122	Message type 17 indicates the tasks have requested to exit.
124, 126, 132, 134	Indicates that the user has closed the STDOUT and STDERR pipes.
127, 129	Message type 15 indicates accounting data.
143-144	Indicates that the home node is sending an exit.
146-147	Indicates that the home node is finding the Job Manager interface.

Appendix B. MPI Safety

This appendix provides information on creating a *safe* MPI program. Much of the information presented here comes from *MPI: A Message-Passing Interface Standard, Version 1.1*, available from the University of Tennesee.

Safe MPI Coding Practices

What's a Safe Program?

This is a hard question to answer. Many people consider a program to be *safe* if no message buffering is 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 is considered to be a conservative programming style, which provides good portability because program completion doesn't depend on the amount of available buffer space.

On the flip side, there are many programmers that prefer more flexibility and use an *unsafe* style that relies, at least somewhat, on buffering. In such cases, the use of standard send operations can provide the best compromise between performance and robustness. Good MPI programs will supply sufficient buffering so that these programs will not result in deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where you want more control. Since buffer overflow conditions are easier to diagnose than deadlock, this mode can also be used for debugging purposes.

Non-blocking message passing operations can be used 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

Message passing programs can hang or deadlock when one task waits for a message that is never sent, or when each task is waiting for the other to send or receive a message. Within a task, a similar situation can occur when one task is waiting for another to release a lock on a shared resource, such as a piece of memory. If the waiting task is holding a lock that is needed by the running task, then both threads will deadlock while waiting for the same lock (mutex).

A more subtle problem 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 thread-safe, because the thread may be pre-empted 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 thread safe */
 {
```

© Copyright IBM Corp. 1997

It is recommended that you don't write threaded message passing programs until you are familiar with writing and debugging threaded, single-task programs.

Note: While the signal handling library (libmpi.a) supports both the MPI standard, as well as MPL (the message passing interface provided by IBM before the MPI standard was adopted), the threaded library (libmpi_r.a) supports only the MPI standard.

Using Threaded Programs with Non-Thread-Safe Libraries: A threaded MPI program must meet the same criteria as any other threaded program; it must avoid using non-thread-safe functions in more than one thread (for example, **strtok**). In addition, it must use only thread-safe libraries, if library functions are called on more than one thread. In AIX, not all the libraries are thread-safe, so you should carefully examine how they are used in your program.

Linking with Libraries Built with libc.a: Compiling a threaded MPI program will cause the **libc_r.a** library to be used to resolve all the calls to the standard C library. If your program links with a library that has been built using the standard C library, it is still usable (assuming that it provides the necessary logical thread safety) under the following conditions:

- The library has been built using a reference to the C library shared object. This is the default, unless the **-bnso** flag was used to build it.
- The runtime library path resolves the file reference libc.a to the POE version of **libc_r.a**.

When your executable is loaded, the loader resolves shared library references using the LIBPATH environment variable first, then the libpath string of the executable itself. POE sets the LIBPATH variable to select the correct message passing library (User Space or IP). The **mpcc_r** (as well as **mpCC_r** and **mpxlf_r**) script sets the libpath string to:

```
/usr/lpp/ppe.poe/lib/threads:/usr/lpp/ppe.poe/lib: ...
```

so that POE versions of **libc.a** and **libc_r.a** will be used. If these libraries are not available, you'll need to set LIBPATH to point to the ones you want to use.

It's important to note that if you have executables that resolve symbols from *both* **libc.a** and **libc_r.a**, your library search path must pick up **libc.a** as a symbolic link to **libc_r.a**, so that a single, consistent version of the library is actually used.

Some General Hints and Tips

To ensure you have a truly MPI-based application, you need to conform to a few basic rules of point-to-point communication. In this section, we'll alert you to some of the things you need to pay attention to as you create your parallel program. Note that most of the information in this section was taken from *MPI: A Message Passing Interface Standard*, so you may want to refer to this document for more information.

Order

With MPI, it's important to know that messages are *non-overtaking*; 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.

Here's an example of using non-overtaking messages. Note that 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
```

Progress

If two processes 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 has already been satisfied by another message, and has itself completed. Likewise, the receive 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's 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's no matching receive. Since the first receive posted by *process 1* doesn't 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
```

Fairness

MPI does not guarantee *fairness* in the way communications are handled, so it's your responsibility to prevent starvation among the operations in your program.

So what might *unfairness* look like? An example might be a situation where a send with a matching receive on another process doesn't 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. A good MPI program uses only a small amount of resources for each pending send and receive, but this 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 because of 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 merely 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 example below 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 other the 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 in order 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 won't complete either, and instead of deadlock, we'll have buffer overflow.

Appendix C. Installation Verification Program Summary

The POE Installation Verification Program (IVP) provides an ideal way to determine if your system is properly set up, prior to running your applications. This appendix contains a summary of what it does. For more detailed information, see *IBM Parallel Environment for AIX: Installation Guide*.

Steps Performed by the POE Installation Verification Program

The Installation Verification Program is located in the **/usr/lpp/ppe.poe/samples/ivp** directory, and is invoked by the **ivp.script** shell script. It checks for the needed files and libraries, and ensures that everything is in order. It also issues messages when it finds something wrong.

You need the following in order to run the **ivp.script**:

- a non-root userid properly authorized in **/etc/hosts.equiv** or the local **.rhosts** file.
- access to a C compiler.

If the conditions above are true, the IVP does the following:

1. Ensures that:
   - MPI library **/usr/lpp/ppe.poe/lib/ip/libmpci.a** is there or linked for IBM RS/6000 SP systems.
   - MPI library **/usr/lpp/ppe.poe/lib/ip/libmpci_r.a** is there or linked for IBM RS/6000 SP systems.
   - MPI library **/usr/lpp/ppe.poe/lib/us/libmpci.a** is there or linked for IBM RS/6000 SP systems.
   - MPI library **/usr/lpp/ppe.poe/lib/us/libmpci_r.a** is there or linked for IBM RS/6000 SP systems.
   - **poe**, **pmdv2**, **mpcc**, and **mpcc_r** are there, and are executable.
   - **mpcc** and **mpcc_r** scripts are in the path.
   - The file **/etc/services** contains an entry for **pmv2**, the Partition Manager daemon.
   - The file **/etc/inetd.conf** contains an entry for **pmv2**, and that the daemon it points to is executable.
2. Creates a work directory in **/tmp/ivp**<**pid**> to compile and run sample programs.

   Note: **pid** is the process id.
3. Compiles sample programs.
4. Creates a **host.list** file with local host names listed twice.
5. Runs sample programs using IP protocol to two tasks, using both threaded and non-threaded libraries.
6. Removes all files from **/tmp**, as well as the temporary directory.
7. Checks for the dbx **bos.adt.debug** fileset, for parallel debuggers.

At the control workstation (or other home node):

© Copyright IBM Corp. 1997

LOGIN as a user other than **root**, and start **ksh**.

ENTER **export LANG=En_C**

ENTER the following:

1. **cd /usr/lpp/ppe.poe/samples/ivp**
2. **ivp.script**

   - This runs an installation verification test that checks for successful execution of a message passing program using two tasks on this node. The output should resemble:

```
Verifying the location of the libraries
Verifying the existence of the Binaries
Partition Manager daemon /etc/pmdv2 is executable
POE files seem to be in order
Compiling the ivp sample program
Output files will be stored in directory /tmp/ivp15480
Creating host.list file for this node
Setting the required environment variables
Executing the parallel program with 2 tasks

POE IVP: running as task 0 on node pe03
POE IVP: running as task 1 on node pe03
POE IVP: there are 2 tasks running
POE IVP: task   1 received <POE IVP Message Passing Text>
POE IVP: all messages sent

Parallel program ivp.out return code was 0

Executing the parallel program with 2 tasks, threaded library

POE IVP_r: running as task 1 on node pe03
POE IVP_r: running as task 0 on node pe03
POE IVP_r: there are 2 tasks running
POE IVP_r: task   1 received <POE IVP Message Passing Text -
           Threaded Library>
POE IVP_r: all messages sent

Parallel program ivp_r.out return code was 0

If both tests return a return code of 0, POE IVP
 is successful.  To test POWERparallel system message passing,
 run the tests in ../samples/poetest.bw and poetest.cast
 To test threaded message passing,
 run the tests in ../samples/threads
End of IVP test
```

If errors are encountered, your output contains messages that describe these errors. You can correct the errors, and run the **ivp.script** again, if desired.

Appendix D. Parallel Environment Internals

This appendix provides some additional information about how the IBM Parallel Environment for AIX (PE) works, with respect to the user's application. Much of this information is also explained in *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*.

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:

- **mpcc**
- **mpcc_r**
- **mpCC**
- **mpCC_r**
- **mpxlf**
- **mpxlf_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() program.
- Signal handlers, for additional control in terminating the program during Visualization Tool (VT) tracing, and enabling the handling of the process termination signals. Appendix G of *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference* explains the signals that are handled in this manner.
- Replacement **exit**(), POE's own version of the **exit**() and **atexit**() functions, in order to synchronize profiling and provide better synchronization upon exit.

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.

If you create a static executable, the application executable and the message passing libraries are statically bound together.

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.

© Copyright IBM Corp. 1997

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 running under the control of its own Partition Manager daemon (pmd). When a parallel job is started, POE contacts the nodes assigned to run the job (called *remote nodes*), and starts a pmd instance for each task. POE sends environment information to the pmd daemon information for the parallel job (including the name of the executable) and the pmd daemon spawns a process to run the executable. The spawned process has standard I/O redirected to socket connections back to the pmd daemon, so any output the application writes to STDOUT or STERR is sent back to the pmd daemon. pmd, in turn, sends the output back to POE via another socket connection and POE writes the output to its STDOUT or STERR. 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 signalling. These capabilities are available to control any parallel program run by POE, and they don't depend on the Message Passing library.

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 installs a signal handler for any of the signals POE supports, it should call the POE registered signal handler if the process decides to terminate. Appendix G of *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference* explains signal handling in greater detail.

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. There are various conditions and values with specific meanings associated with exit status. These are explained in Appendix G of *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*.

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.

Glossary of Terms and Abbreviations

This glossary includes terms and definitions from:

- The *Dictionary of Computing*, New York: McGraw-Hill, 1994.
- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies can be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Definitions are identified by the symbol (A) after the definition.
- The *ANSI/EIA Standard - 440A: Fiber Optic Terminology*, copyright 1989 by the Electronics Industries Association (EIA). Copies can be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue N.W., Washington, D.C. 20006. Definitions are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

This section contains some of the terms that are commonly used in the Parallel Environment books and in this book in particular.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard *Vocabulary for Information Processing* (Copyright 1970 by American National Standards Institute, Incorporated), which was prepared by Subcommittee X3K5 on Terminology and Glossary of the American National Standards Committee X3. ANSI definitions are preceded by an asterisk (*).

Other definitions in this glossary are taken from *IBM Vocabulary for Data Processing, Telecommunications, and Office Systems* (GC20-1699).

© Copyright IBM Corp. 1997

A

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.

AIXwindows Environment/6000. A graphical user interface (GUI) for the RS/6000. It has the following components:

- A graphical user interface and toolkit based on OSF/Motif
- Enhanced X-Windows, an enhanced version of the MIT X Window System
- Graphics Library (GL), a graphical interface library for the applications programmer which is compatible with Silicon Graphics' GL interface.

API. Application Programming Interface.

application. The use to which a data processing system is put; for example, topayroll 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.

B

bandwidth. The total available bit rate of a digital channel.

blocking operation. An operation which 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 in which 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 ANSI standards committee for the C language in 1984 and by Uniforum in 1983.

C++. A general purpose programming language, based on C, which 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.

call arc. The representation of a call between two functions within the Xprofiler function call tree. It appears as a solid line between the two functions. The arrowhead indicates the direction of the call; the function it points to is the one that receives the call. The function making the call is known as the *caller*, while the function receiving the call is known as the *callee*.

chaotic relaxation. An iterative relaxation method which uses a combination of the Gauss-Seidel and Jacobi-Seidel methods. The array of discrete values is divided into sub-regions which can be operated on in parallel. The sub-region boundaries are calculated using Jacobi-Seidel, whereas the sub-region interiors are calculated using Gauss-Seidel. 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. It typically provides excellent price/performance.

collective communication. A communication operation which 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 know as *command aliases*.

Communication Subsystem (CSS). A component of the IBM AIX Parallel System Support Programs that provides software support for the High Performance Switch. It provides two protocols; IP (Internet Protocol) for LAN based communication and US (user space) as

a message passing interface that is optimized for performance over the switch. See also *Internet Protocol* and *User Space*.

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.

control workstation. A workstation attached to the IBM RS/6000 SP SP that serves as a single point of control allowing the administrator or operator to monitor and manage the system using IBM AIX Parallel System Support Programs.

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 severe user error. The current program state is needed for the programmer to diagnose and correct the problem.

core file. A file which 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 either of the PE parallel debuggers, control of the parallel program and the display of its data can be limited to a subset of the tasks that belong 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 the *IBM RS/6000 SP: Command and Technical Reference* (GC23-3900) for details.

domain name. The hierarchical identification of a host system (in a network), consisting of human-readable labels, separated by decimals.

E

Earth. Mostly harmless.

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.

event. An occurrence of significance to a task; for example, the completion of an asynchronous operation such as an input/output operation.

Ethernet. Ethernet is the standard hardware for TCP/IP LANs in the UNIX marketplace. It is a 10 megabit per second baseband type network that uses the contention based CSMA/CD (collision detect) media access method.

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, then no set of circumstances can cause any thread to wait indefinitely for access to the lock.

FDDI. Fiber distributed data interface (100 Mbit/s fiber optic LAN).

file system. In the AIX operating system, the collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

fileset. 1) An individually installable option or update. Options provide specific function while updates correct an error in, or enhance, a previously installed product. 2) One or more separately installable, logically grouped units in an installation package. See also *Licensed Program Product* 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. It's 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.

forty-two (42). 1) The answer to Life! The Universe! And Everything! It was originally determined by the great computer Deep Thought many millions of years ago for a race of hyperintelligent pandimensional beings. 2) The smallest whole number greater than forty-one.

function call tree. A graphical representation of all the functions and calls within an application, which appears in the Xprofiler main window. The functions are represented by green, solid-filled rectangles called function boxes. The size and shape of each function box indicates its CPU usage. Calls between functions are represented by blue arrows, called call arcs, drawn between the function boxes. See also *call arcs*.

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. One divides the program into independent pieces of functionality which are distributed to independent processors. This 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+1st iteration Jacobi-Seidel uses only values calculated in the ith iteration. Gauss-Seidel uses a mixture of values calculated in the ith and i+1st 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, Pascal, Fortran, or COBOL 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**.

GUI (Graphical User Interface). A type of computer interface consisting of a visual metaphor of a real-world scene, often of a desktop. Within that scene are icons, representing actual objects, that the user can access and manipulate with a pointing device.

H

High Performance Switch. The high-performance message passing network, of the IBM RS/6000 SP(SP) machine, that connects all processor nodes.

HIPPI. High performance parallel interface.

hook. **hook** is a **pdbx** command that allows you to re-establish control over all task(s) 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, and providing 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 which 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 Parallel Environment for AIX. A program product that provides an execution and development environment for parallel FORTRAN, C, or C++ programs. It also includes tools for debugging, profiling, and tuning parallel programs.

InfoExplorer. A program that displays hypertext and allows navigation within it.

installation image. A file or collection of files that are required in order to install a software product on an RS/6000 workstation or on SP 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 Product*, and *package*.

Internet. The collection of worldwide networks and gateways which function as a single, cooperative virtual network.

Internet Protocol (IP). 1) The TCP/IP protocol that provides packet delivery between the hardware and user processes. 2) The High Performance Switch library, provided with the IBM AIX Parallel System Support Programs, that follows the IP protocol of TCP/IP.

IP. See *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 which 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 instant at which an instruction control unit initiates a call for data transmission, and the instant at which the actual transfer of data (or receipt of data at the remote end) begins. Latency is related to the hardware characteristics of the system and to the different layers of software that are involved in initiating the task of packing and transmitting the data.

Licensed Program Product (LPP). A collection of software packages, sold as a product, that customers pay for to license. It can consist of packages and filesets a customer would install. These packages and filesets bear a copyright and are offered under the terms and conditions of a licensing agreement. See also *fileset* and *package*.

LoadLeveler. A job management system that allows users to run jobs and match processing needs with system resources, to better utilize 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 which makes multiple copies of the body of a loop, placing the copies also 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

Marketing Division of the Sirius Cybernetics Corporation. A bunch of mindless jerks who'll be the first against the wall when the revolution comes.[1]

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 using the AIX Message Facility 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.

[1] The editors welcome applications from anyone interested in taking over the post of robotics correspondent (this is a joke...get it?).

message passing. Refers to the process by which parallel tasks explicitly exchange program data.

MIMD (Multiple Instruction Multiple Data). A parallel programming model in which different processors perform different instructions on different sets of data.

MPMD (Multiple Program Multiple Data). A parallel programming model in which different, but related, programs are run on different sets of data.

MPI. Message Passing Interface; a standardized API for implementing the message passing model.

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.

node. (1) In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network. (2) In terms of the IBM RS/6000 SP, a single location or workstation in a network. An SP node is a physical entity (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, which returns immediately whether or not the operation was completed. For example, a nonblocking receive will not wait until a message is sent, but a blocking receive will wait. A nonblocking receive will return a status value that indicates whether or not a message was received.

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.

optimization. A not strictly accurate but widely used 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 filesets that have been collected into a single installable image of program products, or LPPs. Multiple filesets can be bundled together for installing groups of software together. See also *fileset* and *Licensed Program Product*.

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) In terms of the IBM RS/6000 SP, a logical definition of nodes to be viewed as one system or domain. System partitioning is a method of organizing the SP 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. **pdbx** is the parallel, symbolic command line debugging facility of PE. **pdbx** is based on the **dbx** debugger and has a similar interface.

PE. The IBM Parallel Environment for AIX program product.

performance monitor. A utility which displays how effectively a system is being used by programs.

POE. See Parallel Operating Environment.

pool. Groups of nodes on an SP that are known to the Resource Manager, and are identified by a number.

point-to-point communication. A communication operation which 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 via 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 which produces an execution profile of an application or program. It is useful to identifying 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.

Program Marker Array. An X-Windows run time monitor tool provided with Parallel Operating Environment, used to provide immediate visual feedback on a program's execution.

pthread. A thread that conforms to the POSIX Threads Programming Model.

R

Ravenous Bugblatter Beast of Traal (RBBT). A creature so mind-bogglingly stupid that it thinks if you can't see it, it can't see you. Ravenous Bugblatter Beasts often make a very good meal for visiting tourists.

reduction operation. An operation, usually mathematical, which reduces a collection of data by one or more dimensions. For example, the arithmetic SUM operation is a reduction operation which reduces an array to a scalar value. Other reduction operations include MAXVAL and MINVAL.

remote host. Any host on a network except the one at which a particular operator is working.

remote shell (rsh). A command supplied with both AIX and the IBM AIX Parallel System Support Programs that lets you issue commands on a remote host.

Report. In Xprofiler, a tabular listing of performance data that is derived from the gmon.out files of an application. There are five types of reports that are generated by Xprofiler, and each one presents different statistical information for an application.

Resource Manager. A server that runs on one of the nodes of an IBM RS/6000 SP (SP) machine. It prevents parallel jobs from interfering with each other, and reports job-related node information. The Resource Manager allows POE applications to request dedicated or shared use of nodes, or the adapters on nodes.

RISC. Reduced Instruction Set Computing (RISC), the technology for today's high performance personal computers and workstations, was invented in 1975.

S

shell script. A sequence of commands that are to be executed by a shell interpreter such as C shell, korn shell, or Bourne shell. Script commands are stored in a file in the same form as if they were typed at a terminal.

segmentation fault. A system-detected error, usually caused by referencing an invalid memory address.

server. A functional unit that provides shared services to workstations over a network; for example, a file server, a print server, a mail server.

signal handling. A type of communication that is used by message passing libraries. Signal handling involves using AIX signals as an asynchronous way to move data in and out of message buffers.

source line. A line of source code.

source code. The input to a compiler or assembler, written in a source language. Contrast with object code.

SP. IBM RS/6000 SP; a scalable system from two to 128 processor nodes, arranged in various physical configurations, that provides a high powered computing environment.

SPMD (Single Program Multiple Data). A parallel programming model in which different processors execute the same program on different sets of data.

standard input (STDIN). In the AIX operating system, 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). In the AIX operating system, 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.

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 may 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.

System Data Repository. A component of the IBM AIX Parallel System Support Programs software that provides configuration management for the SP system. It manages the storage and retrieval of system data across the control workstation, file servers, and nodes.

System Status Array. An X-Windows run time monitor tool, provided with the Parallel Operating Environment, that lets you quickly survey the utilization of processor nodes.

T

task. A unit of computation analogous to an AIX process.

thread. A single, separately dispatchable, unit of execution. There may be one or more threads in a process, and each thread is executed by the operating system concurrently.

tracing. In PE, the collection of data for the Visualization Tool (VT). The program is *traced* by collecting information about the execution of the program in trace records. These records are then accumulated into a trace file which a user visualizes with VT.

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 may not be appropriate). These records are then accumulated into a trace file which allows the Visualization Tool to visually display the communications patterns from the program.

U

unrolling loops. See *loop unrolling*.

US. See *user space*.

user. (1) A person who requires the services of a computing system. (2) Any person or any thing that may issue or receive commands and message to or from the information processing system.

user space (US). A version of the message passing library that is optimized for direct access to the SP High Performance Switch, that maximizes the performance capabilities of the SP hardware.

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.

view. (1) In an information resource directory, the combination of a variation name and revision number

that is used as a component of an access name or of a descriptive name.

Visualization Tool. The PE Visualization Tool. This tool uses information that is captured as your parallel program executes, and presents a graphical display of the program execution. For more information, *see IBM Parallel Environment for AIX: Operation and Use, Vol. 2*.

VT. See *Visualization Tool*.

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.

xpdbx. This is the former name of the PE graphical interface debugging facility, which is now called **pedb**.

Xprofiler. An AIX tool that is used to analyze the performance of both serial and parallel applications, via a graphical user interface. Xprofiler provides quick access to the profiled data, so that the functions that are the most CPU-intensive can be easily identified.

Index

Special Characters

-coredir command 42
-euilib 12
-hostfile option 19
-ilevel option 20
-infolevel option 14, 20, 39, 103
-labelio option 6, 19
-pmdlog option 20
-procs option 6, 19
-rmpool 13
-stdoutmode option 7, 20
.rhosts file 3
&peol.
  communication with nodes 116
  exit status 116
  how it works 115
  internals 115
  signal handling 116
  starting applications 115

Numerics

4-point stencil 73
42, the answer to the meaning of life 35

A

access, to nodes 3
AIX-based
AIX-based user authorization 3
ALLMSG 100
allocation, node
  High Performance Switch 12, 13
  host list file 13
  Resource Manager 12, 13
attach debugger option 55
attaching the debugger 55

B

b_vand 99
b_vor 99
b_vxor 99
babel fish x
bad output
  bad results 63
  error messages 62
bad results 63
Betelguese 1
BROADCAST 95

C

collective communications
  BROADCAST 95
  COMBINE 95
  CONCAT 95
  GATHER 95
  GETLABEL 97
  GETMEMBERS 97
  GETRANK 97
  GETSIZE 97
  GETTASKID 98
  GROUP 98
  INDEX 96
  PARTITION 98
  PREFIX 96
  REDUCE 96
  SCATTER 96
  SHIFT 96
  SYNC 97
COMBINE 95
common problems
  bad output 62
  can't compile a parallel program 37
  can't connect with the remote host 38
  can't execute a parallel program 39
  can't start a parallel job 37
  core dumps 42
  no output 47
  no output or bad output 41
compiler scripts 11
compiling 11
  C example 10
  examples 10
  FORTRAN example 10
  scripts 11
CONCAT 95
constants, global 99
core dumps 42, 47
core files 42
creating MPI objects
  attribute keys 100
  communicators 100
  data types 100
  error handlers 100
  groups 100
  reduction ops 100
  requests 100
  status 100

© Copyright IBM Corp. 1997

D

d_vadd 99
d_vmax 99
d_vmin 99
d_vmul 99
data decomposition 22
debugger, attaching to POE job 55
debugging 63
Dent, Arthur 1
DFS/DCE-based 4
DFS/DCE-based user authorization 4
DONTCARE 100

E

encyclopedia galactica xii
ENVIRON 93
environment variables
  LANG 35
  MP_COREDIR 42
  MP_EUIDEVELOP 62
  MP_EUILIB 12
  MP_HOSTFILE 13, 19, 38
  MP_INFOLEVEL 20, 39, 103
  MP_LABELIO 8, 19, 62
  MP_PMDLOG 20
  MP_PMLIGHTS 65, 66
  MP_PROCS 8, 19, 38, 65
  MP_RESD 38, 40
  MP_RMPOOL 13, 38
  MP_STDOUTMODE 20, 47
  MP_TRACELEVEL 48
  NLSPATH 35
  running POE with 8
error handling 101
error messages 62
errors
  logging to a file 36
examples in this book, getting xiii

F

functional decomposition 29
functions, user-defined 99

G

GATHER 95
GETLABEL 97
GETMEMBERS 97
GETRANK 97
GETSIZE 97
GETTASKID 98
global variables and constants 99
GROUP 98

H

hangs 62
High Performance Switch
  and node allocation 12, 13
host list file 4, 13
host list file, examples 5

I

i_vadd 98
i_vmax 99
i_vmin 99
i_vmul 99
INDEX 96
inetd 39
initialization, how implemented 32
installation 2
Installation Verification Program (IVP) 2, 113
interstellar bypass 67

L

l_vand 99
l_vor 99
LANG 35
Laplace equation 73
last error code 99
linking with libraries built with libc.a
logging errors to a file 36
loops, unrolling
  example 22

M

message length, MPL vs. MPI 100
messages
  and problem determination 35
  finding 36
  format 36
  interpreted 105
  level reported 14, 103
  PE message catalog components 36
  PE message catalog errors 35
  types 105
MP_COREDIR 42
MP_EUIDEVELOP 62
MP_EUILIB 12
MP_HOSTFILE 13, 19
MP_INFOLEVEL 20, 39, 103
MP_LABELIO 8, 19, 62
MP_PMDLOG 20
MP_PMLIGHTS 65, 66
MP_PROCS 8, 19
MP_RESD 40
MP_RMPOOL 13

MP_STDOUTMODE 47
MP_STOUTMODE 20
MP_TRACELEVEL 48
mpc_marker 64, 65
MPI objects, creating 100
MPI to MPL equivalents
  ALLGRP 100
  ALLMSG 100
  DONTCARE 100
  NULLTASK 100
MPI_COMM WORLD 32
MPI_Comm_rank 24
MPI_Comm_size 24
MPI_Finalize 24
MPI_Init 24
MPI_PROD 32
MPI_Reduce 32
MPI_Scan 32
MPI_SUM 32
MPL to MPI equivalents
  b_vand 99
  b_vor 99
  b_vxor 99
  BROADCAST 95
  COMBINE 95
  CONCAT 95
  d_vadd 99
  d_vmax 99
  d_vmin 99
  d_vmul 99
  ENVIRON 93
  GATHER 95
  GETLABEL 97
  GETMEMBERS 97
  GETRANK 97
  GETSIZE 97
  GETTASKID 98
  GROUP 98
  i_vadd 98
  i_vmax 99
  i_vmin 99
  i_vmul 99
  INDEX 96
  l_vand 99
  l_vor 99
  mperrno 99
  PACK 94
  PARTITION 98
  PREFIX 96
  PROBE 95
  RECEIVE (Blocking) 92
  RECEIVE (Non-Blocking) 91
  REDUCE 96
  s_vadd 98
  s_vmax 99
  s_vmin 99
MPL to MPI equivalents *(continued)*
  s_vmul 99
  SCATTER 96
  SEND (Blocking) 91
  SEND (Non-Blocking) 91
  SEND/RECEIVE (Blocking) 92
  SHIFT 96
  STATUS 92
  STOPALL 94
  SYNC 97
  TASK_QUERY 93
  TASK_SET 92
  UNPACK 94
  VRECV 94
  VSEND 94
  WAIT 92
myhosts file 13

N

NLSPATH 35
node allocation
  High Performance Switch 12, 13
  host list file 13
  Resource Manager 12, 13
NULLTASK 100

O

options
  -euilib 12
  -hostfile 19
  -ilevel 20
  -infolevel 14, 20, 103
  -labelio 19
  -pmdlog 20
  -procs 19
  -rmpool 13
  -stdoutmode 20

P

PACK 94
Parallel Operating Environment
  -hostfile option 19
  -ilevel option 20
  -infolevel option 14, 20, 103
  -labelio option 19
  -pmdlog option 20
  -procs option 19
  -stdoutmode option 20
  compiling programs 115
  description 2
  options 19
  running 5
  running, examples 6

Parallel Operating Environment (POE), description 2
parallelizing program 78
PARTITION 98
Partition Manager Daemon 39
pmarray 64
POE
  -euilib 12
  -hostfile option 19
  -ilevel option 20
  -infolevel option 14, 20, 103
  -labelio option 19
  -pmdlog option 20
  -proc option 19
  -rmpool option 13
  -stdoutmode option 20
  communication with nodes 116
  compiling programs 115
  description 2
  exit status 116
  how it works 115
  internals 115
  options 19
  running 5
  running, examples 6
  signal handling 116
  starting applications 115
POE options 19
point-to-point communication
  ENVIRON 93
  PACK 94
  PROBE 95
  RECEIVE (Blocking) 92
  RECEIVE (Non-Blocking) 91
  SEND (Blocking) 91
  SEND (Non-Blocking) 91
  SEND/RECEIVE (Blocking) 92
  STATUS 92
  STOPALL 94
  TASK_QUERY 93
  TASK_SET 92
  UNPACK 94
  VRECV 94
  VSEND 94
  WAIT 92
Prefect, Ford 1
PREFIX 96
PROBE 95
problems, common
  bad output 62
  can't compile a parallel program 37
  can't connect with the remote host 38
  can't execute a parallel program 39
  can't start a parallel job 37
  core dumps 42
  no output 47
  no output or bad output 41
processor node, defined 2
profiling program 74
Program Marker Array
  mpc_marker 64
  pmarray 64
  using 64
protocol implications

R

RECEIVE (Blocking) 92
RECEIVE (Non-Blocking) 91
REDUCE 96
reduction functions
  b_vand 99
  b_vor 99
  b_vxor 99
  d_vadd 99
  d_vmax 99
  d_vmin 99
  d_vmul 99
  i_vadd 98
  i_vmax 99
  i_vmin 99
  i_vmul 99
  l_vand 99
  l_vor 99
  s_vadd 98
  s_vmax 99
  s_vmin 99
  s_vmul 99
remote execution authority, testing 3
running POE
  running 6
  with environment variables 8

S

s_vadd 98
s_vmax 99
s_vmin 99
s_vmul 99
safe coding practices
  fairness 110
  order 109
  resource limitations 110
  safe program, described 107
safety 107
  MPI programs 61
sample program, to illustrate messages 103
sample programs in this book, getting xiii
SCATTER 96
SEND (Blocking) 91
SEND (Non-Blocking) 91
SEND/RECEIVE (Blocking) 92

SHIFT 96
sine series algorithm 29
starting applications with POE 115
startup problems 39
STATUS 92
STOPALL 94
SYNC 97

T

task identifiers, MPL vs. MPI 100
TASK_QUERY 93
TASK_SET 92
threaded programs 11, 34, 47, 63, 72, 107, 108
trademarks vi
tuning 72
  serial algorithm 73
tuning the performance of

U

UNPACK 94
unrolling loops
  example 22
user authorization 3, 4
user-defined functions 99
using with non-thread-safe libraries

V

variables, global 99
Visualization Tool displays, using 52
Visualization Tool, for detecting hangs 61
vogon constructor ship x
VRECV 94
VSEND 94

W

WAIT 92
wildcard receives 100
wildcards 100
wildcards, MPL to MPI equivalents
  ALLGRP 100
  ALLMSG 100
  DONTCARE 100
  NULLTASK 100