IBM AIX Parallel Environment
Operation and Use
Release 2.0

SH26-7230-01

IBM

IBM AIX Parallel Environment
Operation and Use
Release 2.0

SH26-7230-01

Note

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

Second Edition, June 1994

This edition applies to Version 1, Release 2 of IBM AIX Parallel Environment, program number 5765-144, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

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

IBM welcomes your comments. A form for your comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department 52QA, Mail Station 911
1 Enterprise Drive
Kingston, NY 12401-1099
United States of America

FAX: (International Access Code)+1+914+385-0662

IBMLink* (United States customers only): KGNRCF at KGNVMC
IBM Mail Exchange: USIB27BQ at IBMMAIL
Internet: kgnrcf@vnet.ibm.com

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 1993, 1994. 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.. Contains licensed material — Property of IBM.

Contents

Notices vi

About This Book vii
Who Should Use This Book vii
How to Use This Book vii
Overview of Contents vii
Document Conventions viii
Related PE Publications ix
Related AIX Publications x
Other Related Publications x
Ordering Additional Copies of This Book x
Setting POE Environment Variables x
National Language Support xi
Accessing Online Help Information xi

Chapter 1. Introduction 1

Chapter 2. Executing Parallel Programs 7
Executing Parallel Programs Using POE 7
Step 1: Compile the Program 8
Step 2: Copy Files to Individual Nodes 9
Step 3: Set Up the Execution Environment 10
Step 4: Start X-Windows Analysis Tools 25
Step 5: Invoke the Executable 25
Monitoring and Controlling Program Execution 32
Specifying Develop Mode 32
Making POE Wait for Processor Nodes 33
Making POE Ignore Arguments 33
Managing Standard Input, Output, and Error 34
Submitting a POE Job via LoadLeveler 39
Controlling POE Using Parallel Desktop 41
Installing the Parallel Desktop 41
Starting Parallel Desktop 41
Executing Parallel Programs Using Parallel Desktop 43
Monitoring Program Execution Using the Program Marker Array 46
Step 1: Call PM Array Parallel Utility Functions 47
Step 2: Compile the Program 47
Step 3: Set the Number of Lights 47
Step 4: Open the PM Array Window 48
Step 5: Invoke the Program and Monitor its Execution 48
Step 6: Close the PM Array Window 49
Monitoring System Activity Using the System Status Array 50
Opening and Closing the System Status Array Window 51

Chapter 3. Debugging Parallel Programs 53
Getting Started With the pdbx and xpdbx Debuggers 53
Start the Debugger 54
Using the pdbx Debugger 56
Obtaining Information With pdbx Subcommands 58
Setting Command Context and Grouping Tasks 59

© Copyright IBM Corp. 1993

Creating, Removing, and Listing Command Aliases . . . 63
Loading the Partition With the Load Subcommand . . . 64
Using Breakpoints and Tracepoints in pdbx . . . 64
Controlling Execution With pdbx Subcommands . . . 69
Using the xpdbx Debugger . . . 72
Setting the Context and Grouping Tasks . . . 73
Loading the Partition . . . 75
Using the Command Window . . . 76
Controlling Program Execution . . . 77
Examining Program Data . . . 82
Hiding a Task's Data, Stack, and Break/Trace Information . . . 85
Tracing Program Execution . . . 86
Changing the Displayed Source Code . . . 88
Debugging Programs Using Multiple Views . . . 89

Chapter 4. Profiling Parallel Programs . . . 91

Chapter 5. Visualizing Program and System Performance . . . 93
Starting the Visualization Tool . . . 97
Opening and Closing Views . . . 98
Adjusting a View's Display Rate and Colors . . . 101
Saving and Loading a VT Configuration File . . . 103
Using VT for Trace Visualization . . . 105
Types of Trace Records . . . 105
Generating Trace Files . . . 106
Using VT to Play Trace Files . . . 112
Using VT for Performance Monitoring . . . 120
The Performance Monitor Window . . . 120
View Descriptions . . . 124
Communication/Program Views . . . 124
Computation Views . . . 135
Disk Views . . . 146
Network Views . . . 152
System Views . . . 156

Appendix A. Parallel Environment Commands . . . 165
mpcc . . . 166
mpCC . . . 169
mprcp . . . 172
mpxlf . . . 173
pdbx . . . 176
pmarray . . . 194
poe . . . 196
poestat . . . 205
vt . . . 207
xpdbx . . . 210

Appendix B. POE Environment Variables and Command-Line Flags . . . 213

Appendix C. Customizing Resources . . . 219
Program Marker Array Resource Variables . . . 220
VT Resource Variables . . . 220

Appendix D. Visualization Tool Hints and Tips for Application Developers 227
Introduction 227
Tools and Techniques Used in Developing Parallel Applications 230
Machine Setup 230
On-line Performance Monitoring 230
Program Visualization 232
Debugging 233
Using VT in the Development Process 234
Using VT for Trace Visualization 236
Using VT for Performance Monitoring 237
Contents of the VT Graphical Displays 238
Applying VT 245
Visual Understanding of the Program's Operation 245
Analysis and Tuning Considerations 246
Debugging Programs During Development 248
Reducing Trace File Size 249
Understanding How VT Operates 250
Tracing Operation 250
VT Display History 251

Glossary of Terms and Abbreviations 253

Index 257

Notices

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 Corporation
IBM Director of Licensing
208 Harbor Drive
Stamford, Connecticut
United States 06904

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

AIX	POWERparallel
AIXwindows	RISC System/6000
IBM	Scalable POWERparallel Systems
IBMLink	SP1
InfoExplorer	SP2
LoadLeveler	

The following terms, denoted by a double asterisk (**), in this publication, are trademarks of other companies as follows:

AFS	Transarc Corporation
Motif	Open Software Foundation
UNIX	Unix System Laboratories Incorporated
X-Windows System	Massachusetts Institute of Technology

About This Book

This book describes the IBM* AIX* Parallel Environment program product (PE) and how to use its facilities and tools to execute, analyze, debug, and tune parallel programs. It describes the operating environment which enables you to run parallel programs, and also describes the various PE tools for visualizing a program's performance, debugging parallel programs, and tuning parallel programs for their most efficient performance.

This book concentrates on the actual commands, graphical user interfaces, and use of these tools as opposed to the writing of parallel programs. For this reason, you should use this book in conjunction with the *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference*, (SH26-7228).

This book assumes that AIX, X-Windows**, and the PE software are already installed. The PE software is designed to run on a IBM 9076 Scalable POWERparallel System* (SP1* or SP2*), a RISC System/6000* network cluster, or on a mixed system where additional RISC System/6000 processors supplement a POWERparallel* system. For complete information on installing the PE software, see *IBM AIX Parallel Environment: Installation and Diagnosis*, (SH26-7231).

Who Should Use This Book

This book is designed primarily for application developers. It is also intended for end users running parallel programs and some of the information and tools covered should interest system administrators. Readers should have some experience with graphical user interface concepts such as windows, pull-down menus, and menu bars. They should also have knowledge of the AIX operating system and the X-Window system. Where necessary, this book provides some background information relating to these areas. More commonly, this book refers you to the appropriate documentation.

How to Use This Book

Overview of Contents

This book contains the following information:

- Chapter 1, “Introduction” on page 1 is a quick overview of the PE program product. It describes the various PE components, and how you might use each in developing a parallel application program.
- Chapter 2, “Executing Parallel Programs” on page 7 describes how to compile and execute parallel programs using the Parallel Operating Environment (POE). This chapter shows how to execute parallel programs from the AIX command-line prompt, or through Parallel Desktop – the POE AIXwindows* Desktop interface. This chapter also covers the Program Marker Array which allows you to monitor program execution online, and the System Status Array which allows you to monitor system activity online.

© Copyright IBM Corp. 1993

- Chapter 3, “Debugging Parallel Programs” on page 53 describes the two versions of the Parallel Environment's debugging facility – **pdbx** and **xpdbx**. Both debuggers are based on the AIX **dbx** debugger, and both add capabilities specific to parallel programming. The **pdbx** debugger has a line-oriented interface similar to that of the **dbx** debugger. The **xpdbx** debugger uses a Motif** based X-windows interface.
- Chapter 4, “Profiling Parallel Programs” on page 91 describes how to use the AIX profilers **prof** and **gprof** to profile parallel programs.
- Chapter 5, “Visualizing Program and System Performance” on page 93 describes the PE Visualization Tool (VT). Intended for application developers and system administrators, this chapter describes how you can play back statistical and event records – called *trace records* – generated during a program's execution. It also describes how you can use VT to monitor the operational status and activity of each of the processor nodes.
- Appendix A, “Parallel Environment Commands” on page 165 contains the manual pages for the PE commands discussed throughout this book.
- Appendix B, “POE Environment Variables and Command-Line Flags” on page 213 describes the environment variables you can set to influence the execution of parallel programs and the operation of PE tools. This appendix also describes the command-line flags associated with each of the environment variables. When invoking a parallel program, you can use these flags to override the value of an environment variable.
- Appendix C, “Customizing Resources” on page 219 describes how to customize X-Windows resources for PE tools.
- Appendix D, “Visualization Tool Hints and Tips for Application Developers” on page 227 is a separate document included here as an appendix to supplement the information in Chapter 5, “Visualizing Program and System Performance” on page 93. It is a discussion of the VT and how it works with other components of the AIX Parallel Environment. It also includes a discussion of tools and techniques used in the development process, followed by an introduction on how VT fits into the development process. Some of the frequently used VT displays are also explained.

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.

Type Style	Used For
bold	Command names, program names, file names, flag names, directory names, path names, and any text that you must type exactly as it appears.
italic	Parameters and variables. You substitute the text for items shown in italic. For example, when you have to enter a variable called *filename*, you must type the name of the appropriate file in place of *filename*. In addition, italic type is used to introduce new terms, to indicate book titles, and to add emphasis.
`monospace`	Examples and system prompts or responses that appear on the screen.

In addition to these highlighting conventions, this book also uses a number of typographic marks to illustrate command syntax. The following table outlines the meaning of these marks.

Typographic Mark	Description
[ ]	Brackets enclose optional items in syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in syntax descriptions.
|	A vertical bar separates items in a list of choices.
...	Horizontal ellipsis points mean that you can repeat the preceding item one or more times.

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.

Related PE Publications

IBM AIX Parallel Environment: Installation, Administration, and Diagnosis, (SH26-7231).

IBM AIX Parallel Environment: Parallel Programming Subroutine Reference, (SH26-7228)

IBM AIX Parallel Environment: Programming Primer, (SH26-7223)

IBM AIX Parallel Environment: General Information, (GH26-7229)

Related AIX Publications

AIX for RISC System/6000: Commands Reference, (GBOF-1802)

AIX for RISC System/6000: Communications Concepts and Procedures, (GBOF-1524)

AIX for RISC System/6000: Files Reference, (GC23-2200)

AIX for RISC System/6000: General Concepts and Procedures, (GC23-2202)

AIX for RISC System/6000: General Programming Concepts, (SC23-2205)

AIXwindows Environment/6000: AIXwindows and AIXwindows Desktop User's Guide, (GC23-2432).

Other Related Publications

Note: The following is related to this book only if you run parallel programs on a POWERparallel system. It is not related if you use a RISC System/6000 network cluster.

IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide, (SH26-2486)

Note: The following is related to this book only if you run parallel programs using the IBM LoadLeveler* job management system.

IBM LoadLeveler: Users Guide, (SH26-7226)

Ordering Additional Copies of This Book

To order additional copies of this book, use order number SH26-7230.

Setting POE Environment Variables

Throughout this book, a number of POE environment variables are discussed. These environment variables can be set to influence the operation of certain tools and the execution of parallel programs. The POE environment variables also have associated command-line flags that can be used when invoking a parallel program or tool. The POE command-line flags temporarily override their associated environment variable. The following table shows how to set a POE environment variable or command-line flag depending on the shell you are using. The remainder of this book assumes use of the Korn Shell. For a complete listing of the POE environment variables and command-line flags, refer to Appendix B, "POE Environment Variables and Command-Line Flags" on page 213.

	To set an environment variable:	**To override an environment variable when invoking a parallel program or PE tool:**
In Korn Shell:	**ENTER** **export** *VARIABLE=value*	**ENTER** *command program -flag value*
In C Shell	**ENTER** **setenv** *VARIABLE value*	**ENTER** *command program -flag value*

National Language Support

For National Language Support (NLS), all the PE facilities and tools display messages located in externalized message catalogs. An English version of the message catalogs is 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. The **NLSPATH** environment variable specifies a list of directories to search for message catalogs. The directories are searched, in the order listed, to locate and open the message catalog. In resolving the path to the message catalog, **NLSPATH** checks 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=En_US

The PE message catalog is in English, and is located in */usr/lib/nls/msg/En_US*. If your site is using its own translation of the message catalog, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For additional information on NLS and message catalogs, see *IBM AIX Parallel Environment: Installation, Administration, and Diagnosis* and *AIX for RISC System/6000: General Programming Concepts*.

Accessing Online Help Information

The Parallel Environment product provides manual pages as well as online help information in InfoExplorer* format. InfoExplorer is the AIX online information facility. You can use InfoExplorer's extensive search capabilities to find help on any part of the Parallel Environment product. When you press **Help** on one of the product's X-Window's applications, the appropriate book in the Parallel Environment library will appear on the screen in InfoExplorer format. In particular, the X-Windows applications display error messages in their own X-Windows. Pressing **Help** in one of these windows invokes InfoExplorer and opens the appropriate book to the appropriate spot for error resolution.

You can also access the books in InfoExplorer format without using the graphical user interface. To access this book or any other book in this library in InfoExplorer format:

ENTER **info -l pe**

To access a list of manual pages for PE:

ENTER **man pe**

Chapter 1. Introduction

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

The PE is a distributed memory message-passing system. It runs on the RISC System/6000* platform using the AIX operating system (Version 3.2 or later). Specifically, you can use the PE to execute parallel programs on:

- any configuration of the IBM 9076 Scalable POWERparallel System* (SP1* or SP2*) as described in the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*. Essentially, a POWERparallel* system is a collection of RISC System/6000 processors grouped into a number of *frames*. Each frame of a POWERparallel system can contain from eight to 16 RISC System/6000 processors.
- a networked cluster of RISC System/6000 processors.
- a *mixed system*. In a mixed system, additional RISC System/6000 processors supplement the processors of a POWERparallel system.

The RISC System/6000 processors of your system are called *processor nodes*. A parallel program executes as a number of individual, but related, *parallel tasks* on a number of your system's processor nodes. The group of parallel tasks is called a *partition*. The processor nodes are connected on the same LAN, so the parallel tasks of your partition can communicate to exchange data or synchronize execution. If you are using a POWERparallel system:

- Your system may have an optional high-performance switch for communication. The switch increases the speed of communication between nodes. It supports a high volume of message passing with increased bandwidth and low latency.
- Your system administrator can divide its nodes into separate pools. A POWERparallel system pool is a subset of processor nodes and is given an identifying pool number.

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

The remainder of this introduction views the PE from two vantage points. First there is a conceptual drawing of the PE, showing how its various components and tools fit together and interact. Following that is a description of each PE component and tool in relation to a hypothetical application development effort. For the best depth of understanding, refer back to the figure as each PE component and tool is described.

© Copyright IBM Corp. 1993

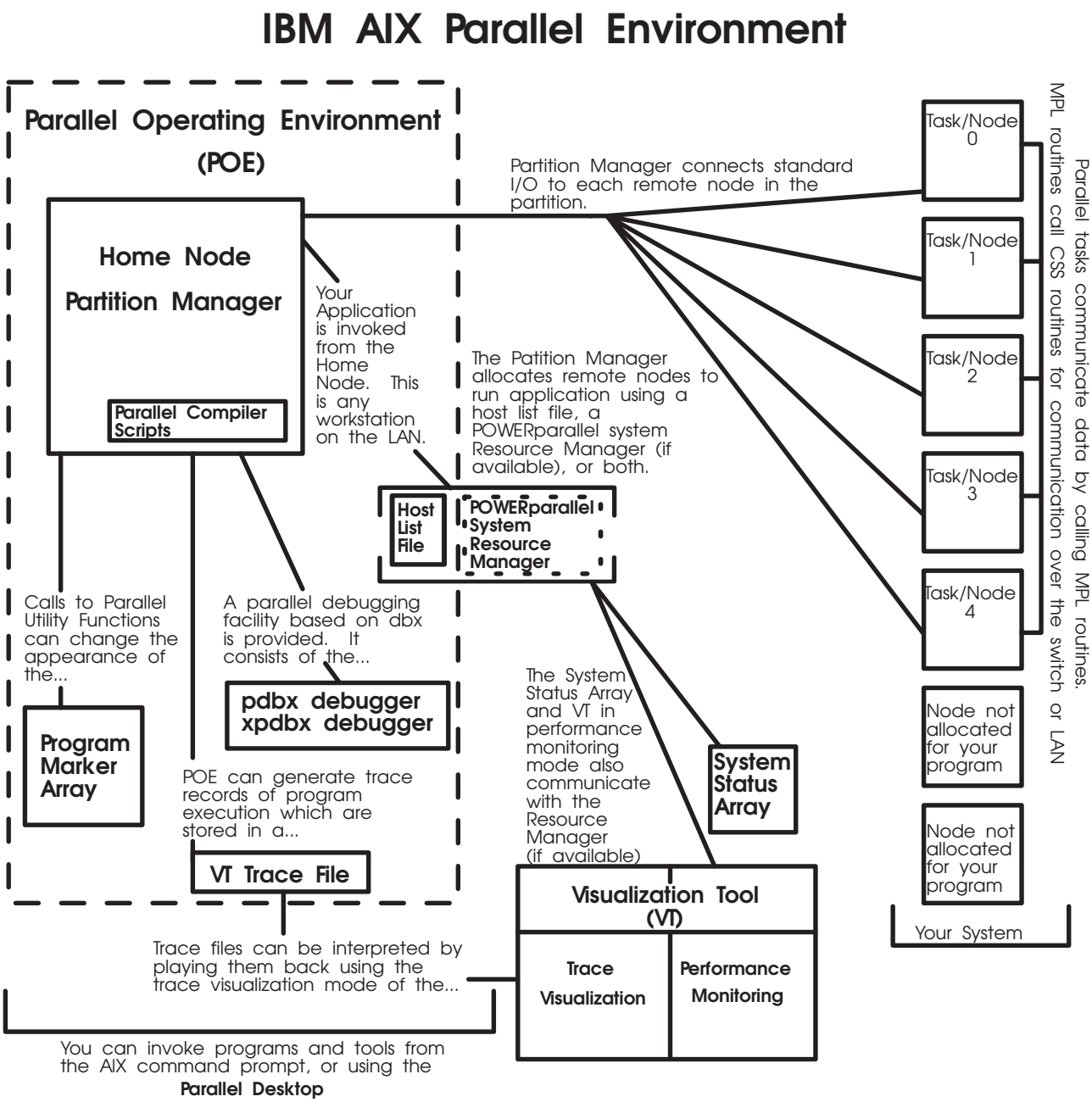


Figure 1. PE Overview

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

The application developer begins by creating a parallel program's source code. The application developer might create this program from scratch or could modify an existing serial program. In either case, the developer places calls to **Message Passing Library (MPL)** routines so that it can run as a number of parallel tasks. This is known as *parallelizing* the application. The MPL calls enable the parallel tasks of your partition to communicate data and coordinate their execution. The MPL routines in turn call Communication Subsystem (CSS) library routines which

handle communication among the processor nodes. There are two separate implementations of the CSS library – the Internet Protocol (IP) CSS and the User Space (US) CSS. While the MPL application interface remains the same, the CSS libraries use different protocols for communication among processor nodes. The IP CSS uses Internet Protocol, while the US CSS is designed for the POWERparallel systems' high-performance switch feature. You can link in the CSS library implementation you want to use either statically (when you compile the parallel program) or dynamically (when you invoke the program). For more information on the MPL subroutine calls, refer to the *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference* and the *IBM AIX Parallel Environment: Programming Primer*.

After writing the parallel program, the application developer then begins a cycle of modification and testing. The application developer now runs his program from his **Home Node** using the **Parallel Operating Environment (POE)**. The Home Node is any workstation on the LAN. POE is an execution environment designed to hide, or at least smooth, the differences between serial and parallel execution. With POE, you invoke a parallel program from your Home Node and run its parallel tasks on a number of **Remote Nodes**. When you invoke a program on your Home Node, POE starts your **Partition Manager** which allocates the nodes of your partition and initializes the local environment. Depending on your hardware and configuration, the Partition Manager uses a **host list file**, the **POWERparallel system Resource Manager**, or both to allocate nodes. A host list file contains an explicit list of node requests, while the Resource Manager allocates nodes from one or more POWERparallel system pools implicitly based on their availability. On a POWERparallel system using the Resource Manager, you can also use a host list file to determine how an allocated node's resources – its adapter and CPU – are used. Your program task can either:

- share or not share the node's adapter
- share or not share the node's CPU.

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

- A number of **parallel compiler scripts**. These are shell scripts that call the six C, C++, or FORTRAN compilers while also linking in an interface library to enable communication between your Home Node and the parallel tasks running on the Remote Nodes. You can statically link in a CSS implementation when you compile a program, or else dynamically link one when you invoke the executable.
- A number of **POE Environment Variables** you can use to set up your execution environment. These are AIX environment variables you can set to influence the operation of POE. These environment variables control such things as how processor nodes are allocated, what programming model you are using, and how standard I/O between the Home Node and the parallel tasks should be handled. Most of the POE environment variables also have

associated command-line flags that enable you to temporarily override the environment variable value when invoking POE and your parallel program.

- Two X-Windows** analysis tools:
  - The **Program Marker Array**. This is a programmable array of small boxes, or *lights*, which are associated with parallel tasks. Under program control, these lights can change color to provide you with immediate visual feedback as your program executes. The Program Marker Array is particularly useful for identifying when a program is hung up, in an endless loop, and so on. See "Monitoring Program Execution Using the Program Marker Array" on page 46 for a complete description of this tool.
  - The **System Status Array**. This tool lets you quickly survey the utilization of your processor nodes. It is useful when listing nodes in a host list file for explicit node allocation, and is discussed in "Monitoring System Activity Using the System Status Array" on page 50.
- The **parallel debugging facility**. This facility, discussed in Chapter 3, "Debugging Parallel Programs" on page 53, is based on the **dbx** debugger, but adds additional functions specific to parallel debugging. The debugging facility has two separate interfaces you can use. The first – **pdbx** – is a line-oriented debugger similar to **dbx**. The other – **xpdbx** – is a Motif**-based X-windows interface.

POE also has an AIXwindows* Desktop interface called **Parallel Desktop**. Just as your default AIXWindows Desktop simplifies working with AIX, Parallel Desktop simplifies working with POE. Source files, executables, and tools are represented on the Parallel Desktop as icons you can easily manipulate to compile and execute parallel programs or start tools.

Once the parallel program is debugged, you now want to tune the program for optimal performance. To do this, you turn to the PE parallel profiling capability and VT to analyze the program.

The **parallel profiling capability** (discussed in Chapter 4, "Profiling Parallel Programs" on page 91) enables you to use the AIX commands **prof** and **gprof** on parallel programs. By using **prof** and **gprof**, you can analyze such things as the execution time spent in each routine, the number of times a routine is called, and the average number of milliseconds per call. The parallel profiling capability enables **prof** and **gprof** to generate multiple profile data files – one for each task.

VT (discussed in Chapter 5, "Visualizing Program and System Performance" on page 93) contains a set of displays which allow you to visualize performance characteristics of your program and system. Each display presents specific, often complex, information in an easily-interpretable form such as a bar chart or a strip graph. You can use VT's displays for *trace visualization* and online *performance monitoring*.

- In *trace visualization*, you play back statistical and event records – or *trace records* – generated by POE during a program's execution. You can use VT to visualize information about the program as well as its use of the underlying system. This visualized information can help you tune the program to optimize its use of the underlying system.
- In *performance monitoring*, you use VT as an online monitor to study the operational status and activity of each of the processor nodes in your

POWERparallel system or RISC System/6000 network cluster. This mode of VT is similar to the System Status Array in that it only displays system statistics and not communication information. Like the System Status Array, it is useful when listing nodes in a host list file for explicit node allocation.

Note:

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

Chapter 2. Executing Parallel Programs

This chapter describes the Parallel Operating Environment (POE). POE is a simple and friendly environment designed to ease the transition from serial to parallel application development and execution. POE lets you develop and run parallel programs using many of the same methods and mechanisms as you would for serial jobs. POE allows you to continue to use the standard UNIX** and AIX application development and execution techniques with which you are already familiar. For example, you can redirect input and output, pipe the output of programs into **more** or **grep**, write shell scripts to invoke parallel programs, and use shell tools such as history. You do all these in just the same way you would for serial programs. So while the concepts and approach to writing parallel programs must necessarily be different, POE makes your working environment as familiar as possible.

This chapter describes the steps involved in compiling and executing your parallel C, C++, or FORTRAN programs using either a POWERparallel system, a RISC System/6000 network cluster, or a mixed system.

Executing Parallel Programs Using POE

This section discusses how to compile and execute your parallel C, C++, or FORTRAN programs. It leaves out the first step in any application's life cycle which is actually writing the program. For information on writing parallel programs, refer to the *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference*, and the *IBM AIX Parallel Environment: Programming Primer*.

In order to execute a parallel program, you need to:

1. Compile and link the program using shell scripts which call the C, C++, or FORTRAN compilers while linking in the Partition Manager interface and MPL subroutines.
2. Copy your executable to the individual nodes in your partition if it is not accessible to the remote nodes.
3. Set up your execution environment. This includes setting the number of tasks, and determining the method of node allocation.
4. Start either of the POE X-Windows analysis tools – the Program Marker Array and the System Status Array – you want to use.
5. Load and execute the parallel program on the processor nodes of your partition. You can:
   - load a copy of the same executable on all nodes of your partition. This is the normal procedure for SPMD programs.
   - individually load the nodes of your partition with separate executables. This is the normal procedure for MPMD programs.
   - load and execute a series of SPMD or MPMD programs, in job step fashion, on all nodes of your partition.

© Copyright IBM Corp. 1993

Step 1: Compile the Program

As with a serial application, you must compile a parallel C, C++, or FORTRAN program before you can run it. Instead of using the **cc**, **xlC**, or **xlf** commands, however, you use the commands **mpcc**, **mpCC**, or **mpxlf**. The **mpcc**, **mpCC**, and **mpxlf** commands not only compile your program, but also link in the Partition Manager and message-passing interface libraries. When you later invoke the program, the subroutines in these libraries enable the Home Node Partition Manager to communicate with the parallel tasks, and the tasks with each other.

These three compiler commands are actually shell scripts which call the **mputil** shell script. The **mputil** shell script in turn calls the appropriate compiler. You can use any of the **cc**, **xlC**, or **xlf** flags on the **mpcc**, **mpCC**, and **mpxlf** commands. In addition, you can use the flags **-ip** or **-us** to statically bind either the IP or US implementation of the communication subsystem (CSS) library with your executable. If you do not use either flag, the CSS library implementation will be dynamically linked when you invoke your program. See “Step 3e: Set the MP_EUILIB Environment Variable” on page 22 for more information.

Use This flag:	To:
-ip	Statically bind the IP CSS library with your executable. This CSS library implementation uses Internet Protocol for communication among processor nodes. If you do not have the POWERparallel systems' high-performance switch feature, you must use the IP CSS.
-us	Statically bind the US CSS library with your executable. This CSS library implementation uses the US protocol for dedicated use of a POWERparallel system's high-performance switch adapter. It allows you to drive the switch adapter directly from your parallel tasks. You can only use the US CSS when running on a POWERparallel system configured with the high-performance switch feature.

The following table shows what to enter to compile a program depending on the language it is written in, and whether or not you want to statically bind in the CSS library implementation for either IP or user space.

To:	Enter:
Compile a C program to statically bind the IP CSS library implementation with the executable.	**mpcc** *program.c* **-ip -o** *program*
Compile a C program to statically bind the US CSS library implementation with the executable.	**mpcc** *program.c* **-us -o** *program*
Compile a C program without binding either CSS library implementation with the executable. A CSS library implementation will be dynamically linked when the executable is invoked.	**mpcc** *program.c* **-o** *program*
Compile a C++ program to statically bind the IP CSS library implementation with the executable.	**mpCC** *program.C* **-ip -o** *program*
Compile a C++ program to statically bind the US CSS library implementation with the executable.	**mpCC** *program.C* **-us -o** *program*
Compile a C++ program without binding either CSS library implementation with the executable. A CSS library implementation will be dynamically linked when the executable is invoked.	**mpCC** *program.C* **-o** *program*
Compile a FORTRAN program to statically bind the IP CSS library implementation with the executable.	**mpxlf** *program.f* **-ip -o** *program*
Compile a FORTRAN program to statically bind the US CSS library implementation with the executable.	**mpxlf** *program.f* **-us -o** *program*
Compile a FORTRAN program without binding either CSS library implementation with the executable. A CSS library implementation will be dynamically linked when the executable is invoked.	**mpxlf** *program.f* **-o** *program*

Note: Be sure to specify the **-g** flag when compiling a program for use with one of the parallel debuggers or with VT. The **-g** flag is a standard compiler flag that produces an object file with symbol table references. This object file is needed by the debuggers and by VT's Source Code view. For more information on the **-g** option, refer to its use on the **cc** command as described in *AIX for RISC System/6000: Commands Reference*. For more information on **mpcc**, **mpCC**, and **mpxlf**, see Appendix A, “Parallel Environment Commands” on page 165.

Step 2: Copy Files to Individual Nodes

Note: You only need to perform this step if your executable, your data files, and (if you plan to use the parallel debuggers) your source code files are not in a commonly accessed, or *shared*, file system. If running on a POWERparallel system, you can also skip this step if the needed files are part of a file collection which is distributed automatically. For information on using file collections, see the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*. For more information on the parallel debuggers, see Chapter 3, “Debugging Parallel Programs” on page 53.

If the program you are running is in a shared file system, the Partition Manager loads a copy of your executable in each processor node in your partition when you invoke a program. If your executable is in a private file system, however, you must copy it to the nodes in your partition. If you plan to use the parallel debuggers **pdbx** and **xpdbx**, you must copy your source files to all nodes as well. You can easily copy files to nodes using the **mprcp** command. All you do is pass **mprcp** the name of the host list file you are using to define your partition and the absolute path name of the file.

For example, to send a copy of *program* to all the processor nodes listed in *host.list* in your current directory:

ENTER **mprcp** *host.list $PWD/program*

▲ The **mprcp** command copies *program* to each of the nodes listed in *host.list* using the **rcp** command. If a program of the same name already exists, the **mprcp** command will overlay it.

For more information on the **rcp** command, refer to *AIX for RISC System/6000: Commands Reference*. For more information on the **mprcp** command, see Appendix A, "Parallel Environment Commands" on page 165.

Step 3: Set Up the Execution Environment

Before invoking your program, you need to set up your execution environment. There are a number of POE environment variables discussed throughout this book and summarized in Appendix B, "POE Environment Variables and Command-Line Flags" on page 213. Any of these environment variables can be set at this time to later influence the execution of parallel programs. This step covers those environment variables most important for successful invocation of a parallel program. When you invoke a parallel program, your home node Partition Manager checks these environment variables to determine:

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

  For *specific node allocation*, the Partition Manager reads an explicit list of nodes contained in a *host list file* you create. If you are using a RISC System/6000 network cluster, or if you are using a mixed system and want to include nodes not on the POWERparallel system, you must use this method of node allocation.

  For *non-specific node allocation*, you give the Partition Manager the number of a POWERparallel system pool, or, using a host list file, a list of POWERparallel system pools. The Partition Manager then connects to the POWERparallel system Resource Manager which allocates nodes from the specified pool(s) for you. You can only use this method for allocation of POWERparallel system nodes.

Note: If you are using a POWERparallel system, and plan to use its high-performance switch adapter for communication, note that each node has been configured by your system administrator for communication using the IP CSS, the US CSS, or both. Any node you request through specific or non-specific node allocation must be configured for the appropriate CSS library implementation. Check with your system administrator to learn which nodes were initialized for the US CSS, and which were initialized for the IP CSS.

There are five separate environment variables that, collectively, determine how nodes are allocated by the Partition Manager. While these are the only ones you must set to allocate nodes, keep in mind that there are many other environment variables you can set. These are summarized in Appendix B, "POE Environment Variables and Command-Line Flags" on page 213, and control such things as

standard I/O handling and VT trace file generation. The environment variables for node allocation are:

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

MP_RESD which specifies whether or not the Partition Manager should connect to the POWERparallel system Resource Manager to allocate nodes.

Note: **MP_RESD** only specifies whether or not to use the POWERparallel system Resource Manager. The actual POWERparallel system Resource Manager you are using is identified by the environment variable **SP_NAME**. The **SP_NAME** environment variable should have been set as part of the POWERparallel system installation process. If it has not, you will need to set **SP_NAME** to the name of the control workstation on the POWERparallel system. For more information on **SP_NAME**, refer to the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*.

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

MP_EUIDEVICE which specifies the adapter set you want to use for IP communication among processor nodes. The Partition Manager only checks this if you are using the IP CSS implementation on the POWERparallel system Resource Manager. It does not check this if you are using a RISC System/6000 network cluster.

MP_RMPOOL which specifies the number of a POWERparallel system pool. The Partition Manager only checks this if you are using the POWERparallel system Resource Manager for nonspecific node allocation without a host list file. You can use the **jm_status** command to return information about POWERparallel system pools. For information on this command, and on POWERparallel system pools in general, refer the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*.

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

For further clarification, the following tables summarize the procedure for determining how nodes are allocated. The tables describe the possible methods of node allocation available to you, what each environment variable must be set to, and whether or not you need to create a host list file. To make the procedure of setting up the execution environment easier and less prone to error, you may eventually wish to create a shell script which automates some of the environment variable settings. To allocate nodes of a POWERparallel system, see Table 1 on page 13. If you are using a RISC System/6000 network cluster, or if you are using a mixed system and want to allocate nodes not on the POWERparallel system, see Table 2 on page 14.

Table 1. Execution Environment Setup Summary (for a POWERparallel system)

	If you want to use the US CSS library for communication among parallel tasks and...		**If you want to use the IP CSS library for communication among parallel tasks and...**	
	you want non-specific node allocation from a single pool:	**you want specific node allocation or if you want non-specific node allocation from more than one pool:**	**you want non-specific node allocation from a single pool:**	**you want specific node allocation or non-specific node allocation from more than one pool:**
MP_PROCS	should be set to the number of program tasks.	should be set to the number of program tasks.	should be set to the number of program tasks.	should be set to the number of program tasks.
A host list file...	is not used.	is used.	is not used.	is used.
MP_HOSTFILE	should be set to an empty string (*""*) or the word "*NULL*"	should be set to the name of your host list file. If not defined, the host list file is assumed to be *host.list* in the current directory.	should be set to an empty string (*""*) or the word "*NULL*"	should be set to the name of your host list file. If not defined, the host list file is assumed to be *host.list* in the current directory.
MP_RESD	should be set to *yes*. If set to an empty string (*""*), or if not set, the Partition Manager assumes **MP_RESD** is *yes*.	should be set to *yes*. If set to an empty string (*""*), or if not set, the Partition Manager assumes **MP_RESD** is *yes*.	should be set to *yes*. If set to an empty string (*""*), or if not set, the Partition Manager assumes **MP_RESD** is *yes*.	should be set to *yes*. If set to an empty string (*""*), the Partition Manager assumes **MP_RESD** is *no*.
MP_EUILIB	*us*	*us*	*ip*	*ip*
MP_EUIDEVICE	should specify the adapter type. Valid, case-sensitive, values are *en0* (Ethernet), *tr0* (token ring), *fi0* (FDDI), and *css0* (the high-performance switch).	should specify the adapter type. Valid, case-sensitive, values are *en0* (Ethernet), *tr0* (token ring), *fi0* (FDDI), and *css0* (the high-performance switch).	should specify the adapter type. Valid, case-sensitive, values are *en0* (Ethernet), *tr0* (token ring), *fi0* (FDDI), and *css0* (the high-performance switch).	should specify the adapter type. Valid, case-sensitive, values are *en0* (Ethernet), *tr0* (token ring), *fi0* (FDDI), and *css0* (the high-performance switch).
MP_RMPOOL	should be set to the number of a POWERparallel system pool.	is validated but not used because you are using a host list file.	should be set to the number of a POWERparallel system pool.	is validated but not used because you are using a host list file.

Table 2. Execution Environment Setup Summary (for RISC System 6000 Network Cluster or Mixed System)

	If you are using a RISC System/6000 network cluster:	**If you are using a mixed system:**
MP_PROCS	should be set to the number of program tasks.	should be set to the number of program tasks.
A host list file...	is used.	is used.
MP_HOSTFILE	should be set to the name of a host list file. If not defined, the host list file is assumed to be *host.list* in the current directory.	should be set to the name of a host list file. If not defined, the host list file is assumed to be *host.list* in the current directory.
MP_RESD	should be set to *no*.	should be set to *yes*. If set to an empty string (*""*), the Partition Manager assumes **MP_RESD** is *no*.
MP_EUILIB	should be set to *ip*.	should be set to *ip*.
MP_EUIDEVICE	is not checked.	should specify the adapter type. Valid, case-sensitive, values are *en0* (Ethernet), *tr0* (token ring), *fi0* (FDDI), and *css0* (the high-performance switch).
MP_RMPOOL	is validated but not used because you are using a host list file.	is validated but not used because you are using a host list file.

Table 3 shows how nodes will be allocated depending on the value of the environment variables discussed in this step. It is provided here for additional illustration. Refer to it in situations when the environment variables are set in patterns other than those recommended in Table 1 on page 13 and Table 2.

Table 3 (Page 1 of 2). Node Allocation Summary

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

Table 3 (Page 2 of 2). Node Allocation Summary

If			Then		
--	--	nodes	Node_List	IP	Nodes
		pools	RM_List	IP	MP_EUIDEVICE
		NULL	RM	US	N/A
	yes	nodes	RM_List	US	N/A
		pools	RM_List	US	N/A
		NULL	RM	US	N/A
	no	nodes	Node_List	IP	Nodes
		pools	Error	IP	Nodes
		NULL	Error	--	--

Note:

Node_List	means that the host list file is used to create the partition.
RM_List	means that the host list file is used to create the partition, but the nodes are requested from the POWERparallel system Resource Manager.
RM	means that the partition is created by requesting nodes in **MP_RMPOOL** from the POWERparallel system Resource Manager.
Nodes	indicates that the external IP address of the processor node is used for communication.
MP_EUIDEVICE	indicates that the IP adapter address indicated by **MP_EUIDEVICE** is used for communication.

Step 3a: Set the MP_PROCS Environment Variable

Before you execute a program, you need to set the size of the partition. To do this, use the **MP_PROCS** environment variable or its associated command-line flag **-procs**. For example, say you want to specify the number of task processes as 6. You could:

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

Invoking parallel programs is discussed in more detail in “Step 5: Invoke the Executable” on page 25.

Notes:

1. Keep in mind that **MP_PROCS** sets the number of task processes and does not necessarily correspond to the number of processor nodes. If tasks are time-sharing processor nodes, for example, the number of tasks will be greater than the number of nodes.
2. If you do not set **MP_PROCS**, the default number of task processes is 1.
3. The examples in this book assume use of the Korn shell. If you are using the C shell, you would have to use the **setenv** command rather than the **export** command. See “Setting POE Environment Variables” on page x for more information.

Step 3b: Create a Host List File

You need to create a host list file if:	You do not need to create a host list file if:
• you are using a RISC System/6000 network cluster. • you are using a mixed system and want to allocate nodes not on the POWERparallel system. • you are using a POWERparallel system and want specific node allocation. • you are using a POWERparallel system and want non-specific node allocation from more than one pool.	you are using a POWERparallel system and want non-specific node allocation from a single pool.

A host list file specifies the processor nodes on which the individual tasks of your program should run. When you invoke a parallel program, your Partition Manager checks to see if you have specified a host list file. If you have, it reads the file to allocate processor nodes. The procedure for creating a host list file differs depending on whether you are using a RISC System/6000 network cluster, a POWERparallel system, or a mixed system. If you are using a RISC System/6000 network cluster, see "Creating a Host List File to Allocate Nodes of a Cluster." If you are using a POWERparallel system or a mixed system, see "Creating a Host List File to Allocate Nodes of a POWERparallel system" on page 17.

Note: If you are using a POWERparallel system, and plan to use its high-performance switch adapter for communication, note that each node has been configured by your system administrator for communication using the IP CSS, the US CSS, or both. Any node you request in the host list file must be configured for the appropriate CSS library implementation. Check with your system administrator to learn which nodes were initialized for the US CSS, and which were initialized for the IP CSS.

Creating a Host List File to Allocate Nodes of a Cluster: If you are using a RISC System/6000, a host list file simply lists a series of host names – one per line. These must be the names of remote nodes accessible from the Home Node. Lines beginning with an exclamation point (!) are comments. The Partition Manager ignores blank lines and comments. The host list file can list more names than are required by the number of program tasks. The additional names are ignored.

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

```
! Host list file for allocating 6 tasks

host1_name
host2_name
host3_name
host4_name
host5_name
host6_name
```

The Partition Manager ignores the first line because it is a comment, and the second line because it is blank. It then allocates *host1_name* to run task 0, *host2_name* to run task 1, *host3_name* to run task 2, and so on. If any of the processor nodes listed in the host list file are unavailable when you invoke your

program, the Partition Manager returns a message stating this and does not run your program.

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

```
host1_name
host2_name
host3_name
host1_name
host2_name
host3_name
```

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

Creating a Host List File to Allocate Nodes of a POWERparallel system: If you are using a POWERparallel system, you can use a host list file for either:

- non-specific node allocation from one or more POWERparallel system pools.
- specific node allocation. If you are using a mixed system and want to allocate nodes not on the POWERparallel system, you must use specific node allocation.

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

For specific node allocation:	For non-specific node allocation from a number of pools:

Each record is either a host name or IP adapter address of a specific processor node of the POWERparallel system. If you are using a mixed system and want to allocate nodes not on the POWERparallel system, you must request them by host name. Lines beginning with an exclamation point are comments. The Partition Manager ignores blank lines and comments.
 	
To understand how the Partition Manager uses a host list file to determine the POWERparallel system nodes on which your program should run, consider the following example host list file:
 	
`! Host list file for allocation 6 tasks`
 	
`host1_name`
`host2_name`
`host3_name`
`host4_adapter_IP_address`
`host5_adapter_IP_address`
`host6_adapter_IP_address`
 	
The Partition Manager ignores the first line because it is a comment, and the second because it is blank. It then allocates *host1_name* to run task 0, *host2_name* to run task 1, *host3_name* to run task 2, and so on. The last three nodes are requested by adapter IP address using dot decimal notation.
 	
Notes:
 	
1. You can also, on each of the records in the host list file, specify how the allocated node's adapter and CPU should be used. For more information, see “Specifying How a Node's Resources Are Used” on page 18.
2. If any of the processor nodes listed in the host list file are unavailable when you invoke your program, the Partition Manager returns a message stating this and does not run your program.
	After installation of a POWERparallel system, your system administrator divides its processor nodes into a number of pools. Each pool has an identifying pool number. For non-specific node allocation from a number of pools, you need to supply the appropriate pool numbers. If you require information about the pools, use the command **jm_status**.
	
	ENTER jm_status -P
	
	▲ The Resource Manager lists information about all POWERparallel system pools.
	
	In a host list file intended for non-specific node allocation from a number of pools, each record is a pool number preceded by an at symbol (@). Lines beginning with an exclamation point are comments. The Partition Manager ignores blank lines and comments.
	
	To understand how the Partition Manager uses a host list file for non-specific node allocation from a number of pools, consider the following example host list file:
	
	`! Host list file for allocating 6 tasks`
	
	`@6`
	`@6`
	`@6`
	`@12`
	`@12`
	`@12`
	
	The Partition Manager ignores the first line because it is a comment, and the second line because it is blank. The at (@) symbols tell the Partition Manager that these are pool requests. It connects to the POWERparallel system Resource Manager to request three nodes from pool 6, and three nodes from pool 12.
	
	Notes:
	
	1. You can also, on each of the records in the host list file, specify how the allocated node's adapter and CPU should be used. For more information, see “Specifying How a Node's Resources Are Used” on page 18.
	2. If there are no nodes available in a requested pool when you invoke your program, the Partition Manager returns a message stating this and does not run your program.
	3. For more information on the **jm_status** command and POWERparallel system pools, refer to the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*.
	4. If the number of program tasks is greater than the number of records in the host list file, the last record in the file is used for the remaining requests.

Specifying How a Node's Resources Are Used: When requesting nodes of a POWERparallel system using a host list file, you can optionally request how each node's resources – its adapter and CPU – should be used. You can specify:

- Whether the node's adapter is to be *dedicated* or *shared*. If dedicated, only a single program task can use it. If *shared*, a number of tasks on that node can use it. If you are using the US CSS library implementation, POE forces the adapter use to be dedicated.
- Whether the node's CPU usage should be *unique* or *multiple*. If *unique*, only your program's tasks can use the CPU. If *multiple*, your program may share the node with other users.

On each record of the host list file, you can make either or both of these specifications. For example, if you wanted your program task to have exclusive use of both the adapter and CPU, the host list record would be:

```
host1_name dedicated unique
```

or

```
host1_name d u
```

This is the same for pool requests:

```
@6 dedicated unique
```

or

```
@6 d u
```

The following tables illustrate how node resources are used. Table 4 shows the default settings for adapter and CPU use, while Table 5 outlines how the two separate specifications determine how the allocated node's resources are used.

Table 4. Adapter/CPU Default Settings

	Adapter	**CPU**
If host list file contains non-specific pool requests:	Dedicated [1]	Unique
If host list file requests specific nodes:	Shared [2]	Multiple

Note:

[1] For IP jobs, adapter is shared.

[2] For US jobs, adapter is dedicated.

Table 5. Adapter/CPU Use

	If the Node's CPU is "Unique":	**If the Node's CPU is "Multiple":**
If the adapter use is "Dedicated":	Intended for production runs of high-performance applications. Only one task uses the adapter and CPU.	The adapter you specified with **MP_EUIDEVICE** is dedicated to your program task. However, you and other users still have access to the CPU through another adapter.
If the adapter use is "Shared":	Only you have access to the node's CPU, but a number of your program's tasks can share the adapter.	Both the adapter and CPU can be used by a number of your program's tasks and other users.

Notes:

1. The US CSS library requires dedicated use of the high-performance switch adapter on the node. If you are using the US CSS for communication among processor nodes, POE forces adapter use to be dedicated. If you are using the US CSS and you specify adapter use to be shared, the specification is ignored.
2. Adapter/CPU usage specification is only enforced for jobs using the POWERparallel system Resource Manager for node allocation.

Generating an Output Host List File: When running parallel programs on a POWERparallel system, you can generate an output host list file of the nodes allocated by the Resource Manager. When you have the Resource Manager perform non-specific node allocation from POWERparallel system pools, this

enables you to learn which nodes were allocated. This information is vital if you want to perform some postmortem analysis or file cleanup on those nodes, or if you want to rerun the program using the same nodes. To generate a host list file, set the **MP_SAVEHOSTFILE** environment variable to a file name. You can specify this using a relative or full path name. As with most POE environment variables, you can temporarily override the value of **MP_SAVEHOSTFILE** using its associated command-line flag **-savehostfile**. For example, to save the Resource Manager's node allocation into a file called */u/hinkle/myhosts*, you could:

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

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

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

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

```
@6
@6
@6
@12
@12
@12
```

The output hostlist file would be:

```
@6
! host1_name
! host1_external_IP_address    host1_adapter_IP_address
@6
! host2_name
! host2_external_IP_address    host2_adapter_IP_address
@6
! host3_name
! host3_external_IP_address    host3_adapter_IP_address
@12
! host4_name
! host4_external_IP_address    host4_adapter_IP_address
@12
! host5_name
! host5_external_IP_address    host5_adapter_IP_address
@12
! host6_name
! host6_external_IP_address    host6_adapter_IP_address
```

Note:

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

Step 3c: Set the MP_HOSTFILE Environment Variable

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

The default host list file used by the Partition Manager to allocate nodes is called *host.list* and is located in your current directory. You can specify a file other than *host.list* by setting the environment variable **MP_HOSTFILE** to the name of a host list file, or by using either the **-hostfile** or **-hfile** flag when invoking the program. In either case, you can specify the file using its relative or full path name. For example, say you want to use the host list file *myhosts* located in the directory */u/hinkle*. You could:

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

If you are using the POWERparallel system Resource Manager for non-specific node allocation from a single pool specified by **MP_RMPOOL**, you must set **MP_HOSTFILE** to an empty string or to the word "*NULL*". Otherwise the Partition Manager will look for a host list file. You can either:

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

Step 3d: Set the MP_RESD Environment Variable

You set the **MP_RESD** environment variable to *yes* or *no* (as specified in Table 1 on page 13 and Table 2 on page 14) to control whether or not the Partition Manager connects to the Resource Manager to allocate processor nodes. If you are allocating nodes of a RISC System/6000 network cluster, you do not have a Resource Manager and should set **MP_RESD** to *no*. If you are using a mixed system, you may set **MP_RESD** to *yes*. However, the Resource Manager only has knowledge of POWERparallel system nodes. To allocate any of the additional RISC System/6000 processors which supplement the POWERparallel system nodes in a mixed system, you must also use a host list file.

As with most POE environment variables, you can temporarily override the value of **MP_RESD** using its associated command-line flag **-resd**. For example, to specify that you want the Partition Manager to connect to the Resource Manager, you could:

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

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

	and you are using a host list file:	and you are not using a host list file:
If MP_EUILIB is set to ip, an empty string, the word "NULL", or if not set:	**MP_RESD** is interpreted as *no* by default, unless host list file includes pool requests.	**MP_RESD** is interpreted as *yes* by default.
If MP_EUILIB is set to us:	**MP_RESD** is interpreted as *yes* by default.	**MP_RESD** is interpreted as *yes* by default.

Note: **MP_RESD** only specifies whether or not to use the POWERparallel system Resource Manager. The actual POWERparallel system Resource Manager you are using is identified by the environment variable **SP_NAME**. The **SP_NAME** environment variable should have been set as part of the POWERparallel system installation process. If it has not, you will need to set **SP_NAME** to the name of the control workstation on the POWERparallel system. For more information on **SP_NAME**, refer to the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*.

Step 3e: Set the MP_EUILIB Environment Variable

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

- *The IP CSS library implementation* uses Internet Protocol for communication among processor nodes. If you do not have the POWERparallel system high-performance switch feature, you must use the IP CSS.
- *The US CSS library implementation* uses the US protocol for dedicated use of a POWERparallel system's high-performance switch adapter. It allows you to drive the switch adapter directly from your parallel tasks. You can only use the US CSS when running on a POWERparallel system configured with the high-performance switch feature.

The **MP_EUILIB** environment variable, or its associated command-line flag **-euilib**, is used to indicate which CSS library implementation you are using. If you have statically bound one of the CSS library implementations with your executable, as described in “Step 1: Compile the Program” on page 8, you have already decided which one you are using. If you have not, POE needs to know which CCL implementation to dynamically link in as part of your executable when you invoke it. The following table shows the appropriate setting for **MP_EUILIB** depending on the CSS library implementation you want and whether or not it has already been statically linked.

	and you have already statically linked it to your executable:	and you want it dynamically linked when you invoke your program:
If you want the IP CSS:	**MP_EUILIB** should either not be set, or else set to an empty string or the word *"NULL"*.	**MP_EUILIB** should be set to *ip*. This specification is case-sensitive.
If you want the US CSS:	**MP_EUILIB** should be set to *us*. This specification is case-sensitive.	**MP_EUILIB** should be set to *us*. This specification is case-sensitive.

For example, say you want to dynamically link in the IP CSS library at execution time. You could:

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

Note:

When you invoke a parallel program, your Partition Manager checks the value of **MP_EUILIB** and then looks to the directory */usr/lpp/poe/lib* for the message-passing library and the CSS library implementation. If you are running on a RISC System/6000 network cluster, this is the actual location of the MPL. If you are running on a POWERparallel system, */usr/lpp/poe/lib* contains symbolic links to the actual location. Consult your system administrator for the actual location of the MPL if necessary.

You can make POE look to a directory other than */usr/lpp/poe/lib* by setting the **MP_EUILIBPATH** environment variable or its associated command-line flag **-euilibpath**. For example, say the CSS library implementations were moved to */usr/altlibs*. To instruct the Partition Manager to look there, you could:

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

The expected library for loading the CSS library implementation is */usr/lpp/poe/lib/***$EUILIB**. Setting the **MP_EUILIBPATH** environment variable causes POE to try to load the CSS library from the directory **$MP_EUILIBPATH/$MP_EUILIB**. If the CSS library (*libcss.a*) is not in the requested path, it will be loaded from the library path for the IP CSS library implementation used when the program was compiled – **$MP_PREFIX***/poe/lib/ip*. **MP_PREFIX** can also be set by the user, but is normally */usr/lpp*. Thus the default library path is normally */usr/lpp/poe/lib/ip*, provided the library is not specified by the **MP_EUILIB** and/or **MP_EUILIBPATH** environment variables.

Step 3f: Set the MP_EUIDEVICE Environment Variable

You need to set the MP_EUIDEVICE environment variable if:	You do not need to set the MP_EUIDEVICE environment variable if:
you have set the **MP_EUILIB** environment variable to *ip*, and are using the POWERparallel system Resource Manager.	you have set the **MP_EUILIB** environment variable to *us*. The Partition Manager assumes that **MP_EUIDEVICE** is *css0* – the high-performance switch adapter.

If you are using the IP CSS library implementation for communication among parallel tasks on a POWERparallel system, you can specify which adapter set to

use for message passing – either Ethernet, FDDI, token-ring, or a POWERparallel system's high-performance switch. The **MP_EUIDEVICE** environment variable and its associated command-line flag **-euidevice** are used to select an alternate adapter set for communication among processor nodes. If neither **MP_EUIDEVICE** device nor the **-euidevice** flag is set, the CSS library uses the external IP address of each remote node. The following table shows the possible, case-sensitive, settings for **MP_EUIDEVICE**.

Setting the MP_EUIDEVICE environment variable to:	Selects:
en0	The Ethernet adapter
fi0	The FDDI adapter
tr0	The token-ring adapter
css0	The POWERparallel systems' high-performance switch adapter

For example, say you want to use IP over the high-performance switch. The nodes have been initialized for IP as described in the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*, and you have already set the **MP_EUILIB** environment variable to *ip*. To specify the high-performance switch, you could:

Set the MP_EUIDEVICE environment variable:	Use the -euidevice flag when invoking the program:
ENTER **export MP_EUIDEVICE=***css0*	**ENTER** **poe** *program* **-euidevice** *css0*

Note: If you do not set the **MP_EUIDEVICE** environment variable, the default is the adapter set used as the external network address.

Step 3g: Set the MP_RMPOOL Environment Variable

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

After installation of a POWERparallel system, your system administrator divides its processor nodes into a number of pools. Each pool has an identifying pool number. If you want non-specific node allocation from a single pool, you need to set the **MP_RMPOOL** environment variable to the number of that pool. If you plan to use the high-performance switch adapter for communication, note that each node has been configured by your system administrator for communication using either the IP CSS, the US CSS, or both. The pool number you specify should consist of nodes configured for the appropriate CSS library implementation. Check with your system administrator to learn which pools consist of nodes initialized for the US CSS and which were initialized for the IP CSS.

If you need information about available pools, use the command **jm_status**.

ENTER **jm_status -P**

▲ The Resource Manager lists information about all POWERparallel system pools.

As with most POE environment variables, you can temporarily override the value of **MP_RMPOOL** using its associated command-line flag **-rmpool**. To specify pool 6, for example, you could:

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

Note: For more information on the **jm_status** command and on POWERparallel system pools, refer to the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*.

Step 4: Start X-Windows Analysis Tools

If you wish to use either of the POE X-Windows analysis tools – the Program Marker Array or the System Status Array – you should start them before invoking the executable. For more information on these tools and how to start them, see "Monitoring Program Execution Using the Program Marker Array" on page 46 and "Monitoring System Activity Using the System Status Array" on page 50.

Step 5: Invoke the Executable

Note: In order to perform this step, you need to have a user account on, and be able to remotely login to, each of the processor nodes. This requires that you have an *.rhosts* file set up in your home directory on each of the remote processor nodes. Alternatively, your user id on the home node can be authorized in the */etc/hosts.equiv* file on each remote node. For more information on the TCP/IP *.rhosts* file format, see *IBM AIX for RISC System/6000: Communications Concepts and Procedures*, and *IBM AIX for RISC System/6000: Files Reference*.

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

- load and execute a SPMD program onto all nodes of your partition. For more information, see "Invoking a SPMD Program" on page 26.
- individually load the nodes of your partition. This capability is intended for MPMD programs. For more information, see "Invoking a MPMD Program" on page 27.
- load and execute a series of SPMD or MPMD programs, in individual job steps, on the same partition. For more information, see "Loading a Series of Programs as Job Steps" on page 29.
- run non-parallel programs on remote nodes. For more information, see "Invoking a Non-Parallel Program On Remote Nodes" on page 31.

When you invoke **poe**, the Partition Manager allocates processor nodes for each task and initializes the local environment. It then loads your program, and reproduces your local environment, on each processor node. The Partition Manager also passes the option list to each remote node. If your program is in a shared file system, the Partition Manager loads a copy of it on each node. If your program is in a private file system, you will have already manually copied your executable to the nodes using the **mprcp** command. If you are using the dynamic message-passing library, the appropriate CSS library implementation (IP CSS or US CSS) is automatically loaded at this time.

Since the Partition Manager attempts to reproduce your local environment on each remote node, your current directory is important. When you invoke **poe**, the Partition Manager will, immediately before running your executable, issue the **cd** command on each remote node. If you are in a local directory that does not exist on remote nodes, you will get an error as the Partition Manager attempts to change to that directory on remote nodes. Typically, this will happen when you invoke **poe** from a directory under */tmp*. We recommend that you invoke **poe** from a file system that is mounted across the system. If it is important that the current directory be under */tmp*, make sure that directory exists on all the remote nodes.

Before using the **poe** command, you must first specify which programming model you are using by setting the **MP_PGMMODEL** environment variable to either *spmd* or *mpmd*. As with most POE environment variables, you can temporarily override the value of **MP_PGMMODEL** using its associated command-line flag **-pgmmodel**. For example, if you want to run a MPMD program, you could:

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

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

Invoking a SPMD Program

If you have a SPMD program, you want to load it as a separate task on each node of your partition. To do this, follow the **poe** command with the program name and any options. The options can be program options or any of the POE command-line flags shown in Appendix B, "POE Environment Variables and Command-Line Flags" on page 213. For compatibility with Release 1.0 of Parallel Environment, you can also invoke a SPMD program by entering the program name and any options. You can also enter **poe** without a program name as described in "Loading Nodes Individually From Standard Input" on page 27. To invoke a SPMD program:

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

or

program [options]

Note: For National Language Support, POE displays messages located in an externalized message catalog. 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=En_US

The PE message catalog is in English, and is located in */usr/lib/nls/msg/En_US*. If your site is using its own translation of the message catalog, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For additional information on NLS and message catalogs, see *IBM AIX Parallel Environment: Installation, Administration, and Diagnosis* and *AIX for RISC System/6000: General Programming Concepts*.

Invoking a MPMD Program

With a SPMD application, a copy of the same executable is sent to, and runs on, each of the processor nodes of your partition. If you are invoking an MPMD application, you are dealing with more than one program and need to individually load the nodes of your partition.

For example, say you have two programs – *master* and *slaves* – designed to run together and communicate via calls to MPL subroutines. The program *master* is designed to run on one processor node. The *slaves* program is designed to run as separate tasks on any number of other nodes. The *master* program will coordinate and synchronize the execution of all the slave tasks. Neither program can run without the other, as *master* only does sends and the *slaves* tasks only do receives.

You can establish a partition and load each node individually using:

- standard input (from the keyboard or redirected)
- a POE commands file

Loading Nodes Individually From Standard Input: To establish a partition and load each node individually using STDIN:

ENTER **poe** *[options]*

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

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

▲ A prompt for the next node in the partition displays.

ENTER the name of the program you want to load on each processor node as you are prompted.

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

For additional illustration, the following shows the command prompts that would appear, as well as the program names you would enter, to load the example *master* and *slaves* programs. This example assumes that the **MP_PROCS** environment variable is set to 5.

```
% poe
0:host1_name> master [options]
1:host2_name> slaves [options]
2:host3_name> slaves [options]
3:host4_name> slaves [options]
4:host5_name> slaves [options]
Partition loaded...
```

Notes:

1. You can use some POE command-line flags on individual program names, but not those that are used to set up the partition. The flags you can use are mainly those having to do with VT trace file collection as described in "Generating Trace Files" on page 106. They are:
   - **-infolevel** or **-ilevel**
   - **-ttempsize** or **-ttsize**
   - **-tmpdir**
   - **-samplefreq** or **-sfreq**
   - **-tbuffwrap** or **-tbwrap**
   - **-tbuffsize** or **-tbsize**
   - **-tpermsize** or **-tpsize**
   - **-euidevelop**
2. By default, the Partition Manager looks to STDIN for the individual program names to load on your partition. However, if the environment variable **MP_CMDFILE** is set to the name of a POE commands file as described in "Loading Nodes Individually Using a POE Commands File," the Partition Manager will look to the commands file instead. To ensure that STDIN is used, check that the **MP_CMDFILE** environment variable is unspecified.
3. You can redirect standard input to the **poe** command. The command will read the various program names from the file you indicate, and will not display any prompts.

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

```
master [options]
slaves [options]
slaves [options]
slaves [options]
slaves [options]
```

Once you have created a POE commands file, you can specify it using a relative or full path name on the **MP_CMDFILE** environment variable or **-cmdfile** flag. For example, if your POE commands file is */u/hinkle/mpmdprog*, you could:

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

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

ENTER **poe** *[options]*

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

Loading a Series of Programs as Job Steps

By default, the Partition Manager releases your partition when your program completes its run. However, you can set the environment variable **MP_NEWJOB**, or its associated command-line flag **-newjob**, to specify that the Partition Manager should maintain your partition for multiple job steps.

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

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

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

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

Note: You can only load a series of programs as job steps using the **poe** command. You can not do this with either of the parallel debugger commands – **pdbx** and **xpdbx** – even though they are built on top of POE.

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

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

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

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

ENTER *setup* [*program-options*]

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

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

ENTER *computation* [*program-options*]

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

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

ENTER *cleanup* [*program-options*]

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

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

ENTER **quit**

or

<ctrl-d>

▲ The Partition Manager releases the nodes of your partition.

Notes:

1. You can also run a series of MPMD programs in job step fashion from STDIN. If **MP_PGMMODEL** is set to *mpmd*, the Partition Manager will, after execution of each step completes, prompt you to individually reload the partition as described in "Loading Nodes Individually From Standard Input" on page 27.
2. When **MP_NEWJOB** is *yes*, the Partition Manager, by default, looks to STDIN for job steps. However, if the environment variable **MP_CMDFILE** is set to the name of a POE commands file as described in "Reading Job Steps From a POE Commands File," the Partition Manger will look to the commands file instead. To ensure that job steps are read from STDIN, check that the **MP_CMDFILE** environment variable is unspecified.

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

```
setup [program-options]
computation [program-options]
cleanup [program-options]
```

If you are loading a series of MPMD programs, the POE commands file is also responsible for individually loading the partition. For example, say you had three master/slave MPMD job steps that you wanted to run as 4 tasks on the same partition. Your POE commands file would contain the following:

```
master1 [options]
slaves1 [options]
slaves1 [options]
slaves1 [options]
master2 [options]
slaves2 [options]
slaves2 [options]
slaves2 [options]
master3 [options]
slaves3 [options]
slaves3 [options]
slaves3 [options]
```

While you could also redirect STDIN to read job steps from a file, a POE commands file gives you more flexibility by not tying up STDIN. You can specify a POE commands file using its relative or full path name. Say your POE commands file is called */u/hinkle/jobsteps*. To specify that the Partition Manager should read job steps from this file rather than STDIN, you could:

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

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

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

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

Invoking a Non-Parallel Program On Remote Nodes

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

ENTER **poe ps**

▲ The process status for each remote node is written to standard out (STDOUT) at your home node. How STDOUT from all the remote nodes is handled at your home node depends on the output mode. See “Managing Standard Output (STDOUT)” on page 35 for more information.

Monitoring and Controlling Program Execution

This section describes a number of additional POE environment variables for monitoring and controlling program execution. It describes how to use the:

- **MP_EUIDEVELOP** environment variable to specify that you want to run your program in MPL develop mode. In this mode, more detailed checking of your program is performed.
- **MP_RETRY** environment variable to make POE wait for processor nodes to become available from the Resource Manager.
- **MP_RETRYCOUNT** environment variable to specify the number of times the Partition Manager should request nodes before returning.
- **MP_NOARGLIST** and **MP_FENCE** environment variable to make POE ignore arguments.
- **MP_STDINMODE** environment variable to manage standard input.
- **MP_STDOUTMODE** environment variable to manage standard output.
- **MP_INFOLEVEL** environment variable to specify the level of messages you want reported to standard error.
- **MP_PMDLOG** environment variable to generate a diagnostic log on remote nodes.
- **MP_LABELIO** environment variable to label message output with task identifiers.

For a complete listing of all POE environment variables, see Appendix B, “POE Environment Variables and Command-Line Flags” on page 213.

Specifying Develop Mode

You can run programs in one of two modes – *develop mode* or *run mode*. In develop mode, intended for developing applications, the MPL performs more detailed checking during execution. Because of the additional checking it performs, develop mode can significantly slow program performance. In run mode, intended for completed applications, only minimal checking is done. While run mode is the default, you can use the **MP_EUIDEVELOP** environment variable to specify MPL develop mode. As with most POE environment variables, **MP_EUIDEVELOP** has an associated command-line flag **-euidevelop**. To specify MPL develop mode, you could:

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

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

Note: You can also specify the mode from within your application. This is done by calling the MP_TASK_SET or mpc_task_set subroutine. If set from within your application program, **MP_EUIDEVELOP** is overridden. For more information, refer to *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference*.

Making POE Wait for Processor Nodes

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

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

The **MP_RETRY** environment variable, and its associated command-line flag **-retry**, specifies the interval (in seconds) to wait before repeating the node request. The **MP_RETRYCOUNT** environment variable, and its associated command-line flag **-retrycount**, specifies the number of times the Partition Manager should make the request before returning. For example, if you wanted to retry the node request five times at five minute (300 second) intervals, you could:

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

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

Making POE Ignore Arguments

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

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

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

ENTER **export MP_NOARGLIST=***yes*

When the **MP_NOARGLIST** environment variable is set to *yes*, POE does not examine the argument list at all. It simply passes the entire list on to the program. For this reason, you can not use any POE command-line flags, but must use the

POE environment variables exclusively. While most POE environment variables have associated command-line flags, **MP_NOARGLIST**, for obvious reasons, does not. To specify that POE should again examine argument lists, either set **MP_NOARGLIST** to *no*, or unset it.

ENTER **export MP_NOARGLIST=***no*

or

unset MP_NOARGLIST

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

ENTER **export MP_FENCE=***Q*

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

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

Managing Standard Input, Output, and Error

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

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

Managing Standard Input (STDIN)

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

- all tasks should receive the same input data from STDIN. This is *multiple input mode*.

- STDIN should be sent to a single task of your partition. This is *single input mode*.

Multiple Input Mode: Setting **MP_STDINMODE** to *all* indicates that all tasks should receive the same input data from STDIN. The Home Node Partition Manager sends STDIN to each task as it is read.

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

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

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

Single Input Mode: There are times when you only want a single task to read from STDIN. To do this, you set **MP_STDINMODE** to the appropriate task id. For example, say you have a MPMD application consisting of two programs – *master* and *slaves*. The program *master* is designed to run as a single task on one processor node. The *slaves* program is designed to run as separate tasks on any number of other nodes. The *master* program handles all I/O, so only its task needs to read STDIN. If *master* is running as task 0, you need to specify that only task 0 should receive STDIN. To do this, you could:

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

Managing Standard Output (STDOUT)

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

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

Unordered Output Mode: Setting **MP_STDOUTMODE** to *unordered* specifies that all tasks should write output data to STDOUT asynchronously. To specify unordered output mode, you could:

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

Notes:

1. If you do not set the **MP_STDOUTMODE** environment variable or use the **-stdoutmode** command-line flag, unordered output mode is the default.
2. If you are using unordered output mode, you will probably want the messages labeled by task id. Otherwise it will be difficult to know which task sent which message. See "Labeling Message Output" on page 37 for more information.
3. You can also specify unordered output mode from your program by calling the MP_STDOUTMODE or mpc_stdoutmode Parallel Utility Function. Refer to *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference* for more information.

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

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

To specify ordered output mode, you could:

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

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

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

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

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

Labeling Message Output

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

```
  7: Hello World
  0: Hello World
  3: Hello World
 23: Hello World
 14: Hello World
  9: Hello World
```

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

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

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

Setting the Message Reporting Level for Standard Error (STDERR)

You can set the environment variable **MP_INFOLEVEL** to specify the level of messages you want from POE. You can set the value of **MP_INFOLEVEL** to one of the integers shown in the following table. The integers *0*, *1*, and *2* give you different levels of informational, warning, and error messages. The integers *3* through *5* indicate debug levels that provide additional debugging and diagnostic information. Should you require help from the IBM Support Center in resolving a PE-related problem, you will probably be asked to run with one of the debug levels. As with most POE environment variables, you can override **MP_INFOLEVEL** when you invoke a program. This is done using either the **-infolevel** or **-ilevel** flag followed by the appropriate integer.

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

Let us say you want the POE message level set to verbose. The following table shows the two ways to do this. You could:

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

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

Generating a Diagnostic Log on Remote Nodes

Using the **MP_PMDLOG** environment variable, you can also specify that diagnostic messages should be logged to a file in */tmp* on each of the remote nodes of your partition. The log file is named mplog.*pid*, where *pid* is the AIX process id of the program task. Should you require help from the IBM Support Center in resolving a PE-related problem, you will probably be asked to generate these diagnostic logs.

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

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

Note: By default, **MP_PMDLOG** is set to *no*. No diagnostic logs are generated.

Submitting a POE Job via LoadLeveler

This brief section is only intended for users who wish to submit POE jobs via IBM LoadLeveller. Refer to *IBM LoadLeveler: User's Guide* for more information on using this job management system.

To submit a POE job via LoadLeveler, you will need to build a LoadLeveler job file which specifies the number of nodes to be allocated, any POE options passed via environment variables, and the name of your executable. The nodes must be known to the POWERparallel system Resource Manager, and must be in a single batch pool.

For example, to run the program *myprog* on five nodes, using a Token Ring adapter for IP message passing, with the message level set to *2* for Verbose messages, the job file would be:

```
#
# @ input = job.in
# @ output = job.out
# @ error = job.err
# @ parallelmachines = 5
# @ checkpoint = no
# @ environment = COPY_ALL; \
                  MP_INFOLEVEL=2; \
                  MP_EUILIB=ip;  \
                  MP_EUIDEVICE=tr0;
# @ arguments = myprogram myarg1 myarg2
    poe
```

The arguments *myarg1* and *myarg2* are to be passed to *myprog*.

To run *myprog* on 32 nodes, using the US CSS with the message level set to *0* for error messages only, the job file would be:

```
#
# @ input = job.in
# @ output = job.out
# @ error = job.err
# @ parallelmachines = 32
# @ checkpoint = no
# @ environment = COPY_ALL; \
                  MP_INFOLEVEL=0; \
                  MP_EUILIB=us;  \
                  MP_HOSTFILE=NULL; \
                  MP_EUIDEVICE=css0;
# @ arguments = myprogram myarg1 myarg2
    poe
```

For compatibility with Parallel Environment Release 1, you may alternatively use the name of your program as the executable. The previous example in this compatibility format would be:

```
#
# @ input = job.in
# @ output = job.out
# @ error = job.err
# @ parallelmachines = 32
# @ checkpoint = no
# @ environment = COPY_ALL; \
                  MP_INFOLEVEL=0; \
                  MP_EUILIB=us;  \
                  MP_HOSTFILE=NULL; \
                  MP_EUIDEVICE=css0;
# @ arguments = myarg1 myarg2
    myprogram
```

Notes:

1. The **MP_EUILIB** environment variable must be set if the POE dynamically linked message passing library support is being used.
2. When setting the environment string, make sure that no white space characters follow the backslash.

Controlling POE Using Parallel Desktop

Parallel Desktop is the POE AIXwindows Desktop interface. AIXwindows Desktop is a graphical user interface between you and the operating system, and is described in *AIXwindows Environment/6000: AIXwindows and AIXwindows Desktop User's Guide*, and *RISC System/6000: Getting Started: Using RISC System/6000*. Before reading this section, you should be familiar with how to use AIXwindows Desktop as described in either of these books.

The Parallel Desktop is a subdesktop of whatever desktop you normally use – either the General desktop, the Power desktop, or the Admin desktop. Just as your default desktop simplifies working with AIX, Parallel Desktop simplifies working with POE. Source files, executables, and PE tools are represented on the Parallel Desktop as icons you can easily manipulate to compile and execute parallel programs or to start the PE tools.

As with all desktops, you can tailor Parallel Desktop by editing its Rule file. Rule files determine aspects of a desktop's appearance, behavior, and actions. Refer to *AIXwindows Environment/6000: AIXwindows and AIXwindows Desktop User's Guide* for information on editing Rule files.

This section describes how to:

- install the Parallel Desktop
- start the Parallel Desktop
- execute parallel programs using the Parallel Desktop

Uses of Parallel Desktop that pertain to other PE tools such as the Program Marker Array or the Visualization Tool are covered in the section or chapter on the tool.

Installing the Parallel Desktop

To install the Parallel desktop into your environment, you need to:

- copy */usr/lpp/poe/samples/desktop/xdtparallel* to *.xdtparallel* in your home directory. From your home directory:

  ENTER **cp /usr/lpp/poe/samples/desktop/xdtparallel .xdtparallel**

- copy */usr/lpp/poe/samples/desktop/XDesktop3* to *XDesktop3* in your home directory. From your home directory:

  ENTER **cp /usr/lpp/poe/samples/desktop/XDesktop3 XDesktop3**

Starting Parallel Desktop

Once you have installed the Parallel Desktop as described in “Installing the Parallel Desktop,” it will automatically start when you start the AIXwindows Desktop program. Depending on how your system is set up, AIXwindows Desktop may automatically start when you login. If it does not start automatically for you:

ENTER **xdt &**

▲ The AIXwindows Desktop starts running as a background process. A short greeting displays, and your default main desktop – ether the General Desktop, the Power Desktop, or the Admin Desktop – opens. The Parallel Desktop program starts and a dialog box appears, prompting you to “Enter IP or US for message passing library”. Here you decide which CSS library

implementation you wish to use – the IP CSS or the US CSS. The IP CSS uses the Internet Protocol for communication among processor nodes. The US CSS uses the US protocol for dedicated use of a POWERparallel system's high-performance switch adapter. It lets you drive the switch adapter directly from you parallel tasks, and can only be used when running on a POWERparallel system configured with the high-performance switch feature.

FOCUS on the text-entry field of the dialog box.

For communication using IP protocol:	For communication using US protocol:
TYPE IN IP ▲ POE uses the IP CSS library implementation for communication among processor nodes. If you do not have the POWERparallel systems' high-performance switch feature, you must use the IP CSS.	**TYPE IN US** ▲ POE uses the US CSS library implementation for communication among processor nodes. You can only use the US CSS when running on a POWERparallel system configured with the high-performance switch feature.

PRESS the **OK** push button on the dialog window.

▲ The dialog window closes and the Parallel Desktop window opens.

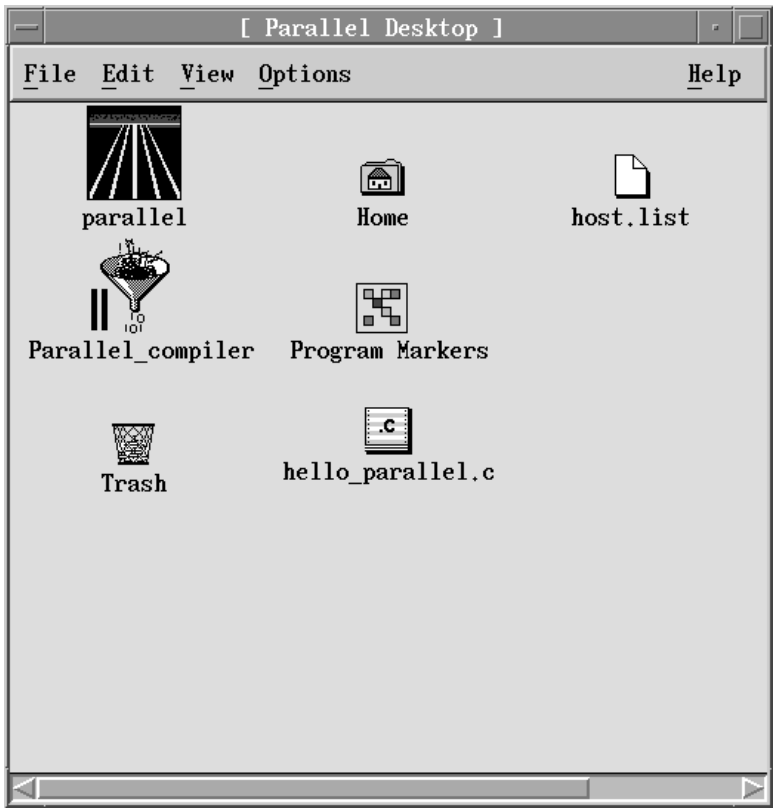


Figure 2. Parallel Desktop window

As initialized, the Parallel Desktop window contains the icons described in the following table.

An icon representing:	Actual file or directory name:
your home directory	$HOME
the parallel executables directory	$HOME/parallel
the parallel compiler (the same icon represents any of the three scripts.)	/usr/lpp/poe/bin/mpcc /usr/lpp/poe/bin/mpCC or /usr/lpp/poe/bin/mpxlf
the Program Marker Array	/usr/lpp/poe/bin/pmarray
the System Status Array	/usr/lpp/poe/bin/poestat
the parallel environment variables	$HOME/parallel/parallel.env
the Visualization Tool (if installed)	/usr/bin/vt
a sample program source	$HOME/parallel/hello_parallel.c (copied from /usr/lpp/poe/samples/marker/hello_parallel.c)
a sample host list file	$HOME/parallel/host.list

Executing Parallel Programs Using Parallel Desktop

Step 1: Create a Host List File

If you are using a POWERparallel system and want non-specific node allocation from a single pool, skip this step. If you are using IP, you need to create a host list file before compiling and invoking your program. When executing a parallel program using the POE, you can create the host list file after you compile your program. However, when using the Parallel Desktop to execute parallel programs, you must create the host list file *before* you compile the program. This is because, after compiling your executable, the Parallel Desktop automatically copies it to each of the processor nodes listed in *host.list*.

To edit *host.list*:

DOUBLE-CLICK on the host list file icon with the left mouse button.

▲ A text editor starts and the host list file opens. By default, the text editor is **vi**. For more information on exactly what needs to go into a host list file, see “Step 3b: Create a Host List File” on page 16.

Step 2: Compile the Program

To compile a parallel program using the Parallel Desktop:

DRAG and DROP the program's icon onto the compiler icon using the left mouse button. The program name must end with an **.o**, **.f**, **.c** or **.C** extension.

▲ The executable is created in the directory *$HOME/parallel*. When you opened the Parallel Desktop window, you were prompted to “Enter IP or US for message passing library”. If you entered **IP**, then the IP CSS library implementation is bound with your executable, and a copy of your executable is sent to each processor node listed in *host.list*.

If you entered **US** when prompted for the message-passing library, then the US CSS is bound with the executable.

If you want to specify compile flags when compiling a parallel program using the Parallel Desktop:

DRAG and DROP the program's icon onto the compiler icon using the right mouse button.

▲ The Compile Flags dialog window opens.

FOCUS on the text entry field of the dialog window.

TYPE IN a string of options. This can be any of the regular **cc**, **xlC**, or **xlf** compiler options.

PRESS **OK**

▲ The string of options is passed to the appropriate compiler.

Step 3: Set Up the Execution Environment

Before invoking your program, your execution environment needs to be set up. There are a number of POE environment variables discussed throughout this book and summarized in Appendix B, "POE Environment Variables and Command-Line Flags" on page 213. The environment variables most important for successful invocation of a parallel program are discussed in "Step 3: Set Up the Execution Environment" on page 10. At the very least, you will need to specify the size of the partition by setting the **MP_PROCS** environment variable equal to the number of program tasks. If **MP_PROCS** is the only environment variable you need to set, you can either set it now, or else skip this step and set it when you invoke a parallel executable.

To set the MP_PROCS environment variable using the Parallel Desktop:		**To set any POE environment variable using Parallel Desktop:**	
DOUBLE-CLICK	on the parallel environment variables icon with the left mouse button.	**DOUBLE-CLICK**	on the parallel environment variables icon with the right mouse button.
	▲ The **MP_PROCS** dialog window opens, prompting you for the number of task processes.		▲ A text editor starts, and opens a script showing the settings of POE environment variables. By default, the text editor is **vi**.
FOCUS	on the text entry field of the dialog window	**EDIT**	the file, changing the POE environment variable settings as desired.
TYPE IN	the number of processes you want your program to run as.		
PRESS	**OK**		
	▲ The dialog window closes and **MP_PROCS** is set to the number you specified. If you are using the IP CSS, and the **MP_RESD** environment variable is not set, then a second dialog window appears asking if you want to use the Resource Manager.		
PRESS	**Yes** to set the **MP_RESD** environment variable to *yes*, or **No** to set the **MP_RESD** environment variable to *no*.		
	▲ The dialog window closes and the **MP_RESD** environment variable is set to either *yes* or *no* as indicated.		

Step 4: Invoke the Executable

Note: If you wish to use either of the POE X-Window analysis tools – the Program Marker Array or the System Status Array – you should start them before performing this step. See "Monitoring Program Execution Using the Program Marker Array" on page 46 and "Monitoring System Activity Using the System Status Array" on page 50 for more information.

Once you have compiled your program, you can invoke the executable. If the executable's icon is not yet on the Parallel Desktop:

DOUBLE-CLICK on the icon representing the parallel directory.

▲ The parallel directory window opens. This window displays a separate icon for each of the executable files you have compiled using Parallel Desktop.

POSITION the Parallel Desktop and parallel directory windows so you can see the contents of each.

DRAG the icon representing the executable you wish to run from the directory window over to the Parallel Desktop.

Once its icon is on the Parallel Desktop, you can invoke the parallel executable. How you do this depends on whether or not you have already set the **MP_PROCS** environment variable as described in the previous step.

If you have already set the MP_PROCS environment variable as described in "Step 3: Set Up the Execution Environment" on page 44:	**If you have not yet set the MP_PROCS environment variable:**
DRAG and DROP the executable's icon onto the parallel environment icon using the left mouse button. ▲ Your program begins executing on the processor nodes of your partition.	**DOUBLE-CLICK** on the executable's icon using the left mouse button. ▲ The **MP_PROCS** dialog window opens, prompting you for the number of task processes. **FOCUS** on the text entry field of the dialog window. **TYPE IN** the number of processes you want your program to run as. **PRESS** **OK** ▲ The dialog window closes, **MP_PROCS** is set to the number you specified, and your program begins executing on the nodes of your partition.

Note: You can specify command-line options for the executable by manipulating its icon in the above steps using the right mouse button instead of the left. These can be program options or POE command-line options as described in Appendix B, "POE Environment Variables and Command-Line Flags" on page 213. If you **DRAG and DROP** or **DOUBLE-CLICK** the executable's icon in the above directions using the right mouse button, an Options dialog window opens. To specify options:

FOCUS on the text entry field of the dialog window.

TYPE IN a string of options.

PRESS **OK**

▲ The dialog window closes and the string of options are appended to the command-line of the program being run.

Monitoring Program Execution Using the Program Marker Array

The Program Marker Array (shown in Figure 3) is an X-Windows run-time analysis tool. This window consists of a number of small squares called *lights* that change color under program control. Each task in a parallel program has its own row of lights, and Parallel Utility Function calls from those tasks can change light colors. The calls can also send strings to the PM Array.

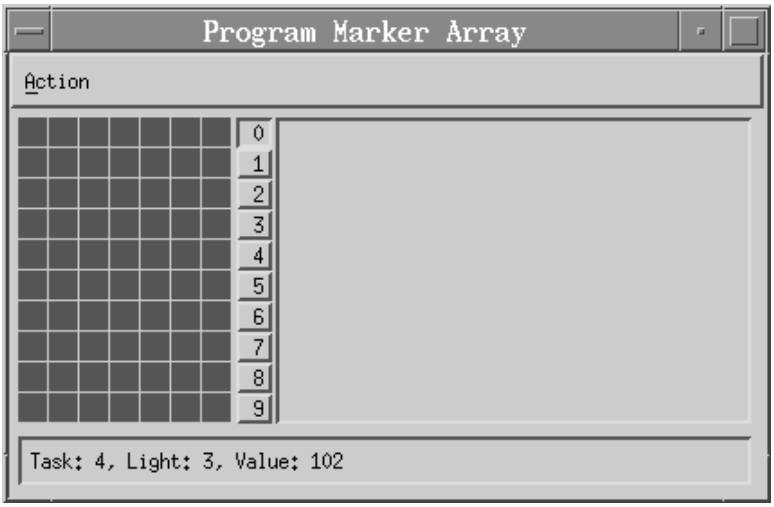


Figure 3. The Program Marker Array

The ability to color lights on, and send strings to, the PM Array window enables a parallel program to provide you with immediate visual feedback as it executes. A program could begin by coloring lights red and then slowly move through the spectrum towards blue as it executes. If a program takes a long time to run, this would give you an indication that it was indeed progressing. Should the program not be progressing, the PM Array would indicate that as well. For example, lights "stuck" on a particular color could indicate that the program is stuck as well. The strings displayed could provide additional information on the program's progress. In addition, the Program Marker Array is distributed as source code, so you can customize the program as you see fit. The source code is located in the directory */usr/lpp/poe/samples/marker*.

In order to use the PM Array to monitor program execution, you need to:

1. In your C, C++, and/or FORTRAN program, insert subroutine calls to color lights on, and send strings to, the PM Array. This is described in "Step 1: Call PM Array Parallel Utility Functions" on page 47.
2. Compile and link the program using the **mpcc**, **mpCC**, or **mpxlf** command. These commands call the C, C++, or FORTRAN compilers while linking in the Partition Manager interface and Parallel Utility subroutines.
3. Set the environment variable **MP_PMLIGHTS** equal to the number of lights you would like displayed per program task. Alternatively, you can set the number of lights using a command-line flag when invoking your parallel program. See "Step 3: Set the Number of Lights" on page 47.
4. Issue the **pmarray** command to start the PM Array program as described in "Step 4: Open the PM Array Window" on page 48.

5. Invoke your parallel program and monitor its execution using the PM Array. This is described in “Step 5: Invoke the Program and Monitor its Execution” on page 48.

Step 1: Call PM Array Parallel Utility Functions

In order for the PM Array to display meaningful information at run time, you need to place calls to Parallel Utility Functions within your program. At run time, your program can then:

- color lights on, and/or send output strings to, the PM Array Window. This is done by calling (in C programs) the mpc_marker or (in FORTRAN programs) the MP_MARKER Parallel Utility Function.
- determine the number of lights displayed per task row. This is done by calling (in C programs) the mpc_nlights or (in FORTRAN programs) the MP_NLIGHTS Parallel Utility Function. Since the number of lights displayed for each task on the PM Array can vary from run to run, this capability is important. It enables your program to learn exactly how many lights are available to be set. It returns an integer value that can then be used by the program to resolve some conditional expression.

The syntax of these Parallel Utility Functions is shown in *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference*.

Step 2: Compile the Program

Once you have inserted calls to the mpc_marker and mpc_nlights functions or the MP_MARKER and MP_NLIGHTS subroutines into your program, you can compile it. Since this is the same procedure you follow when regularly compiling a parallel program with POE, see page 8 for more information.

Step 3: Set the Number of Lights

When you open the PM Array window in the next step, the number of rows in the PM Array are set to the number of program tasks – the current setting of **MP_PROCS**. You can also specify the number of lights you want displayed per task row. To do this, set the environment variable **MP_PMLIGHTS** or specify the **-pmlights** command-line flag in“Step 5: Invoke the Program and Monitor its Execution” on page 48 .

For example, say you want five lights displayed per task in the PM Array. You could:

Set the MP_PMLIGHTS environment variable:	**Use the -pmlights flag when invoking your executable:**
ENTER **export MP_PMLIGHTS=***5*	**ENTER** **poe** *program* **-pmlights** *5*

As with most POE command-line flags, the **-pmlights** flag temporarily overrides its associated environment variable.

Notes:

1. Setting the **MP_PMLIGHTS** environment variable, or the **-pmlights** flag, to *0* indicates that you do not want your program to communicate with the PM Array tool.
2. If you reset the **MP_PMLIGHTS** environment variable, or the **-pmlights** flag, after the Program Marker Array tool is started, it will usually reset to the new number of lights. The only time it will not, however, is when the new value of **MP_PMLIGHTS** is *0*.

Step 4: Open the PM Array Window

The **pmarray** command starts the PM Array program. You will probably want to use the **&** operator so the program runs in the background and does not tie up the aixterm window.

ENTER **pmarray &**

▲ The PM Array window opens. The number of task rows displayed in the PM Array is equal to the current setting of **MP_PROCS**. The number of lights per task row is determined by the current setting of **MP_PMLIGHTS**. If, when you invoke your program in "Step 5: Invoke the Program and Monitor its Execution," you override either of the environment variables using its associated command-line flag, the PM Array redisplays with the new number of rows and/or lights.

Note: The PM Array connects to the Partition Manager using a socket assigned, by default, to port 9999. If you get an error message indicating that the port is in use, specify a different port by setting the **MP_USRPORT** environment variable before entering the **pmarray** command. For example, to specify port 9998:

ENTER **export MP_USRPORT=***9998*

Opening the PM Array Window Using Parallel Desktop

You can also start the PM Array Program and open the PM Array Window using the Parallel Desktop. To do this:

DOUBLE-CLICK on the PM Array icon using the left mouse button.

▲ The PM Array window opens.

Step 5: Invoke the Program and Monitor its Execution

Finally, you invoke your program. As the program runs, the Parallel Utility Function calls placed within it change the color of lights on the PM Array. With appropriate mouse clicks on this window, you can:

- display details of a light
- display output strings from a task
- close the window and discontinue monitoring

Displaying Details of a Light

Each light on the PM Array is associated with a particular task, has a particular light number, and has a particular color value. You can display these details for each of the lights on the PM Array.

For example, say you have coded the PM Array subroutines into your program so that the lights slowly move during execution through the spectrum over color values 0 to 99. As the program runs, the lights start off black, and then turn brown, green, blue, and so on. By watching the lights as they change color, you get a general idea of the program's progress. For a more precise indication of the program's progress, you could display the actual color value number for a light. In this example, the closer this light's value is to 99, the closer execution is to being complete.

To display details of a light:

PLACE the cursor over any light on the PM Array.

PRESS the left mouse button.

▲ The following information displays in the text area at the bottom of the PM Array window:

- the task identifier number
- the light number
- the color value number

This information is not updated until you select another light.

Displaying Task Output

You can display output strings sent by the tasks of your program in the output display area of the PM Array window. This is the area to the right of the PM Array, and the strings displayed there are the ones you specified on the mpc_marker or MP_MARKER subroutine calls. Only one task's strings are displayed in this area at a time. By default, output from task 0 is displayed. You can select the task and display its output instead by pressing its task push button. Each task has a push button. It is just to the right of the task's row on the PM Array, and is labeled with the task identifier. To select, for example, task 3:

PRESS the task pushbutton labeled 3.

▲ Output strings from task 3 are displayed in the output display area. Only one string is displayed at a time.

Note: If a task not currently selected has sent new output to the PM Array window, its task push button will appear yellow.

Step 6: Close the PM Array Window

The PM Array window remains open after your parallel program completes executing. You could then repeat “Step 5: Invoke the Program and Monitor its Execution” to monitor the same, or a different program's execution. To close the PM Array window when you are done monitoring:

SELECT **Action → Quit**

Monitoring System Activity Using the System Status Array

The System Status Array is an X-Windows analysis tool that lets you quickly survey the utilization of processor nodes. This tool is particularly useful if you are not using the POWERparallel system Resource Manager, and so must manually schedule which nodes should be used to run a parallel program. The System Status Array lets you easily see the CPU utilization of each of your processor nodes. Using a host list file, you can then have your program run on those processor nodes you expect to be the least busy.

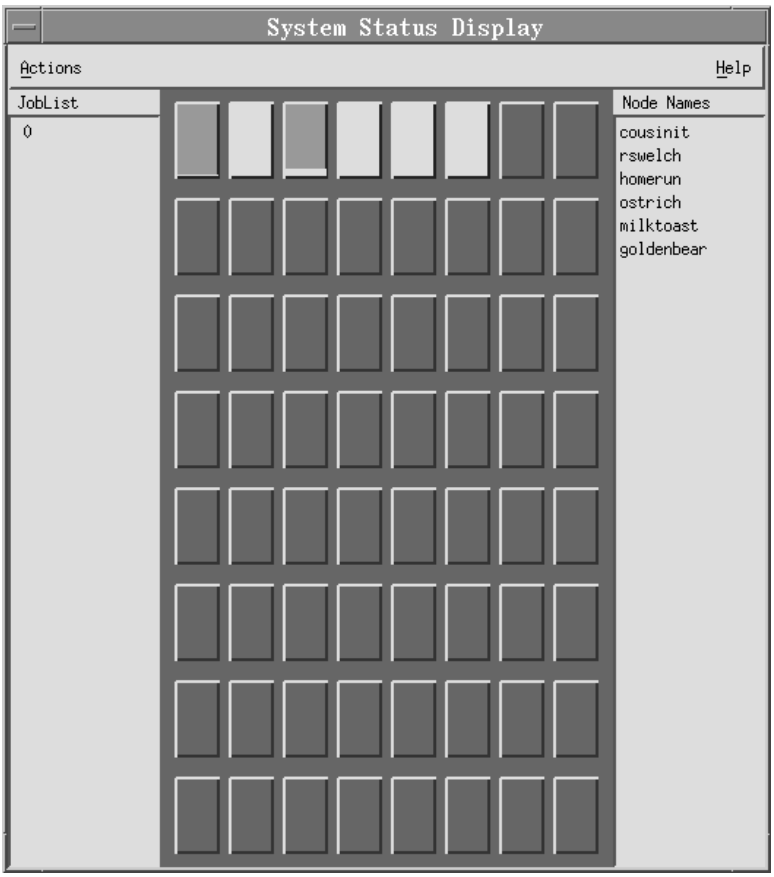


Figure 4. The System Status Array

Each square on the System Status Array represents a processor node of your POWERparallel system or cluster. The squares are colored pink and yellow to show the instantaneous percent of CPU utilization for each processor node. If a square were to appear all pink, it would be at 0 percent utilization. If a square were to appear all yellow, it would be at 100 percent utilization. If a square appears gray, the node is unavailable for monitoring – either it does not have the Statistics Collector daemon (*dig*) running, or the System Status Array cannot communicate with it.

To the right of the Array is a node list which contains the name of each node in the Array. You use this list to identify the name of a node represented in the Array. The nodes are listed in order, left to right, starting with the top row of the Array. If you are using a POWERparallel system, the nodes are displayed in the pool order returned by the **jm_status** command. For more information on this command, see the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*. If you

are using a RISC System/6000 network cluster, the order of nodes displayed is determined by the order in which they are contacted.

Notes:

1. In order to use this tool, the Visualization Tool statistics collector daemon process (*dig*) needs to be running on each of the nodes you wish to monitor. This daemon feeds the System Status Array with the CPU information it displays, and is created on each of your nodes as part of the Visualization Tool's installation procedure. For this reason, VT must be installed in order to use the System Status Array. See *IBM AIX Parallel Environment: Installation, Administration, and Diagnosis* for more information on installing VT.
2. In order to monitor a POWERparallel system, the Resource Manager must be active and available. For information on installing and starting the Resource Manager daemon, refer to the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*.

Opening and Closing the System Status Array Window

The **poestat** command starts the System Status Array, and opens its window. How you invoke this command differs depending on whether you are monitoring a POWERparallel system or a RISC System/6000 network cluster. Also different is the method the Array employs to find the processor nodes to monitor. In either case, you will probably want to use the **&** operator so the program runs in the background and does not tie up the aixterm window.

If you are monitoring a POWERparallel system:	**If you are monitoring a RISC System/6000 network cluster:**
First make sure the environment variable **SP_NAME** is set to the name of your control workstation. This environment variable identifies the POWERparallel system Resource Manager you are using, and is described in the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*. **ENTER** **export SP_NAME=***control_workstation_name* To start the Array: **ENTER** **poestat &** ▲ The System Status Array window (shown in Figure 4 on page 50) opens. The System Status Array program connects to the Resource Manager to automatically select all the nodes of your POWERparallel system for monitoring.	**ENTER** **poestat -norm &** ▲ The System Status Array window (shown in Figure 4 on page 50) opens. The System Status Array program selects for monitoring each of the nodes on the LAN that have the Statistics Collector Daemon running. It also selects each of the nodes listed in the host list file indicated by the **MP_HOSTFILE** environment variable. If **MP_HOSTFILE** is not set, the default is *host.list* in your current directory. **Note:** If a node listed in the host list file references a node that is also on the LAN, the node will be represented twice on the Array.

Note:

For National Language Support, this X-Windows tool displays messages located in an externalized message catalog. If you get an error saying that a message catalog is not found, and want the default message catalog that we provide:

ENTER **export NLSPATH=/usr/lib/nls/msg/%L/%N**

export LANG=En_US

The PE message catalog is in English, and is located in */usr/lib/nls/msg/En_US*. If your site is using its own translation of the

message catalog, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For additional information on NLS and message catalogs, see *IBM AIX Parallel Environment: Installation, Administration, and Diagnosis* and *AIX for RISC System/6000: General Programming Concepts*.

When the System Status Array opens, all nodes display status. You can selectively stop and start displaying status for a single node or all nodes.

To toggle between displaying and not displaying status for a single node:

PLACE the cursor over the name of the processor node.

PRESS the left mouse button.

To toggle between displaying and not displaying status for all nodes:

PLACE the cursor over the digit that appears in the column to the left of the Array under the word “JobList”.

PRESS the left mouse button.

To close the System Status Array window once you have finished monitoring:

SELECT **Actions → Done**

Chapter 3. Debugging Parallel Programs

This chapter describes the Parallel Environment's debugging facility, and how to debug parallel programs using either of its debuggers – **pdbx** or **xpdbx**. Both run under POE and are based, and build, on the AIX **dbx** debugger. The **pdbx** and **xpdbx** debuggers are POE applications that run as a server task on the Home Node and as a number of separate **dbx** client tasks on the remote nodes. The **dbx** client tasks in turn run the parallel tasks of your application for debugging. The **pdbx** or **xpdbx** server running on the Home Node acts as a command interpreter to manage communication between you and the various **dbx** clients. The difference is that **pdbx** maintains the line-oriented interface of **dbx**, while **xpdbx** uses a Motif-based X-windows interface. Those not as familiar with **dbx** may prefer **xpdbx**.

If you are well versed in **dbx** and prefer staying with its line-oriented interface, use **pdbx**. The **pdbx** debugger supports most of the familiar **dbx** subcommands, as well as additional **pdbx** subcommands. Refer to the online manual page for **pdbx** to get a listing of its subcommands. This manual page also appears in Appendix A, “Parallel Environment Commands” on page 165.

This chapter begins with a section describing how to get started with either the **pdbx** or **xpdbx** debugger. The remainder of this chapter is divided into two sections describing each of the debuggers in turn.

For more information on **dbx**, consult *AIX for RISC System/6000: General Programming Concepts* and *AIX for RISC System/6000: Commands Reference*.

Getting Started With the pdbx and xpdbx Debuggers

Before invoking a parallel program using either **pdbx** or **xpdbx** for interactive debugging, you first need to compile the program and set up the execution environment as you would to invoke a parallel program with the **poe** command. The **pdbx** and **xpdbx** debuggers run in the Parallel Operating Environment, so the steps prior to invoking a program are the same. See:

- “Step 1: Compile the Program” on page 8. Be sure to specify the **-g** flag when compiling the program. This produces an object file with symbol table references needed for symbolic debugging. For more information on the **-g** option, refer to its use on other profiler commands such as **cc**, **xlC**, **xlf**. These profiler commands are described in the *AIX for RISC System/6000: Commands Reference*, or your on-line manual pages.
- “Step 2: Copy Files to Individual Nodes” on page 9. You only need to perform this step if your executable and its source code are not in a commonly-accessed file system. If you are running on a POWERparallel system, you can also skip this step if the necessary files are part of a file collection which is distributed automatically.
- “Step 3: Set Up the Execution Environment” on page 10.

As you read these steps, keep in mind that the **pdbx** and **xpdbx** commands for starting the debuggers accept all the option flags that **poe** accepts, and respond to the same environment variables.

© Copyright IBM Corp. 1993

Start the Debugger

Before using the parallel debugging facility, you must decide which of the debuggers you want to use – either **pdbx** or **xpdbx**. The **pdbx** debugger is a line-oriented interface, while the **xpdbx** debugger is a Motif-based X-windows interface.

To invoke the executable and start the parallel debugging facility, use either the **pdbx** or **xpdbx** command. If you are debugging a SPMD program, you can just enter its name on the **pdbx** or **xpdbx** command. It will be loaded on all the nodes of your partition. If you are debugging a MPMD program, you will individually load the nodes of your partition after the debugger is started.

ENTER **pdbx** [*program*] [*program-options*] [*POE-options*] [*pdbx-options*]

or

xpdbx [*program*] [*program-options*] [*POE-options*] [*xpdbx-options*] [*X-Windows-options*]

▲ This starts the specified debugger for interactive debugging. If you specified a *program*, it is loaded on each node of your partition.

Note:

For National Language Support, both **pdbx** and **xpdbx** display messages located in an externalized message catalog. 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=En_US

The PE message catalog is in English, and is located in */usr/lib/nls/msg/En_US*. If your site is using its own translation of the message catalog, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For additional information on NLS and message catalogs, see *IBM AIX Parallel Environment: Installation, Administration, and Diagnosis* and *AIX for RISC System/6000: General Programming Concepts*.

If you started the **pdbx** debugger, see “Using the pdbx Debugger” on page 56. For a listing of **pdbx** subcommands, you can also refer to its online manual page. This manual page also appears in Appendix A, “Parallel Environment Commands” on page 165

If you started the **xpdbx** debugger, see “Using the xpdbx Debugger” on page 72.

The options you specify on the **pdbx** and **xpdbx** commands can be program options, POE options, or one of the debugging options listed in Table 6 on page 55. You can also specify X-Windows options on the **xpdbx** command. You can use the same command-line flags on the **pdbx** and **xpdbx** commands as you use when invoking a parallel program using the **poe** command. For example, you can override the **MP_PROCS** variable by specifying the number of processes with the **-procs** flag. Or you could use the **-hostfile** flag to specify the name of a host list file. For more information on the POE command-line flags, see Appendix B, “POE Environment Variables and Command-Line Flags” on page 213.

The **xpdbx** command accepts standard X-Windows flags. You can also use several common **dbx** command-line flags on the **pdbx** and **xpdbx** commands as shown in the following table.

Table 6. Debugger Option Flags

Use this flag:	To:	For example:
-c	Read **pdbx** startup commands from a specified file. This flag only applies to the **pdbx** command, and cannot be used on the **xpdbx** command. The commands stored in the specified file are executed before command input is accepted from the keyboard. If the **-c** flag is not used, the **pdbx** debug program attempts to read startup commands from the file *.pdbxinit*. To find this file, it first looks in the current directory, and then in the user's home directory. In a **pdbx** session, you can also read commands from a file using the **source** subcommand. "Reading Subcommands From a Commands File" on page 69 describes how to use this subcommand in **pdbx**.	To start the **pdbx** debugger and read startup commands from a file called *start.cmd*: **ENTER** **pdbx -c** *start.cmd*
-f	Read only required symbol table information upon initiation of **dbx**. This flag is passed directly to **dbx**. You can use it to speed startup and to cut down memory usage. This is especially useful when debugging large executable files and when page space is low.	To start the **pdbx** or **xpdbx** debugger and read only required symbol table information: **ENTER** **pdbx -f** or **xpdbx -f**
-I (upper case i)	Add one or more directories to the list of directories searched for an executable's source files. This is important because, if a source file for an executable is not contained in any of the directories on this list, the debugger is not able show the source code. The list of directories searched can also be replaced on a group or single-node basis. If you are using the **pdbx** debugger, you can modify the list of directories with the **use** subcommand. For more information on the **use** subcommand of the **dbx** command, refer to the *AIX for RISC System/6000: Commands Reference*. If you are using the **xpdbx** debugger, you can modify the list of directories using the File Selection Dialog window described in "Opening a Source Code File" on page 88.	To add *directory1* to the list of directories to be searched when starting the **pdbx** or **xpdbx** debugger: **ENTER** **pdbx -I** *directory1* or **xpdbx -I** *directory1* You can add as many directories as you like to the directory list in this way. For example, to add two directories: **ENTER** **pdbx -I** *directory1* **-I** *directory2* or **xpdbx -I** *directory1* **-I** *directory2*
-d	Sets the limit for the nesting of program blocks. The default nesting depth limit is 25. This flag is passed to **dbx** unmodified.	To specify a nesting depth limit: **ENTER** **pdbx -d** *nesting.depth* or **xpdbx -d** *nesting.depth*
-t	Increases the size of the type table for each **dbx** session by the number of types specified by the Number variable. The -t flag is needed when an executable file contains more than 5000 types.	To increase the size of the type table: **ENTER** **pdbx -t** *number* or **xpdbx -t** *number*
-x	Prevents the dbx command from stripping _ (trailing underscore) characters from symbols originating in FORTRAN source code. This flag allows dbx to distinguish between symbols which are identical except for an underscore character, such as xxx and xxx_.	To prevent trailing underscores from being stripped from symbols in FORTRAN source code: **ENTER** **pdbx -x** or **xpdbx -x**

For more information on the option flags described in this table, refer to their use with the AIX command **dbx** as described in *AIX for RISC System/6000: Commands Reference* and *AIX for RISC System/6000: General Programming Concepts*.

Using the pdbx Debugger

This section describes the **pdbx** debugger. This debugger maintains the **dbx** debugger's line-oriented interface and subcommands. Some of these subcommands, however, have been modified for use on parallel programs. After **pdbx** initializes, the **pdbx** command prompt displays to mark the start of a **pdbx** session.

It is at this prompt that you enter **pdbx** subcommands. The debugger supports most of the familiar **dbx** subcommands, as well as additional **pdbx** subcommands. When using the **pdbx** debugger, you should keep in mind that **pdbx** subcommands can either be *context sensitive* or *context insensitive*.

In **pdbx**, *command context* refers to a setting that controls which task(s) receive the subcommands entered at the **pdbx** command prompt. A default command context is provided which contains all tasks in your partition. You can, however, set the command context on a single task or a group of tasks as described in "Setting Command Context and Grouping Tasks" on page 59. Context sensitive subcommands only affect those tasks in the current command context. The debugger directs context sensitive subcommands to just the tasks in the current command context. Context insensitive subcommands are not affected by the command context setting. These include subcommands for setting the command context, accessing help and other information, and ending a **pdbx** session.

The following table outlines the **pdbx** subcommands described in this chapter. It contains both the context insensitive and context sensitive subcommands. Complete syntax information for all these subcommands is also provided under the entry for the **pdbx** command in Appendix A, "Parallel Environment Commands" on page 165.

In addition to the new **pdbx** subcommands described in this table, you can use most of the **dbx** commands. If you are already familiar with **dbx**, you should be aware that some of the **dbx** subcommands behave somewhat differently in **pdbx**. Be aware that:

- all the **dbx** subcommands are context sensitive in **pdbx**. If you use the **stop** subcommand, for example, it will only set breakpoints in the tasks in the current context.
- redirection from **dbx** subcommands is not supported.
- you cannot use the **dbx** subcommands **clear**, **detach**, **edit**, **multproc**, **prompt**, **run**, **rerun**, and **screen**.
- since the application being debugged runs in the Parallel Operating Environment, output from the parallel tasks may not be ordered. You can force task ordering, however, using the **MP_STDOUTMODE** environment variable or the **-stdoutmode** flag when invoking your program with **pdbx**. For more information, see "Managing Standard Output (STDOUT)" on page 35. You can also specify ordered output from within your application using the MP_STDOUTMODE Parallel Utility Function. Refer to the *IBM AIX Parallel*

Environment: Parallel Programming Subroutine Reference for more information.

Context Insensitive pdbx Subcommands		
This subcommand:	**Is used to:**	**For more information see:**
active	Display the list of active tasks. A task that is active is able to receive commands from the debugger.	page 59
alias	Specify a command alias.	page 63
dhelp	Display a brief listing of common **dbx** subcommands.	page 59
group	Group individual tasks under a common name for easier manipulation.	page 60
help	Display a brief listing of **pdbx** subcommands.	page 58
on	Set the command context used to direct subsequent commands to a specific task or group of tasks. This subcommand can also be used to deviate from the command context for a single command without changing the current command context.	pages 59 and 62
quit	End a **pdbx** session.	page 71
source	read **pdbx** subcommands from a specified file.	page 69
unalias	Remove a command alias specified by the **alias** subcommand.	page 63
Context Sensitive pdbx Subcommands		
This Subcommand:	**Is used to:**	**For more information see:**
delete	Remove breakpoints and tracepoints set by the **stop** and **trace** subcommands.	page 69
dbx	Issue a **dbx** subcommand directly to the **dbx** sessions running on the remote nodes. This subcommand is not intended for casual use. It must be used with caution, because it circumvents the **pdbx** server which normally manages communication between the user and the remote **dbx** sessions. It enables experienced **dbx** users to communicate directly with remote **dbx** sessions, but can cause problems as **pdbx** will have no knowledge of the communication that transpired.	the online PE manual page for **pdbx**. This manual page also appears in Appendix A, "Parallel Environment Commands" on page 165.
hook	Regain control over an unhooked task.	page 70
load	load a program on each node in the current context.	page 64
status	Display a list of breakpoints and tracepoints set by the **stop** and **trace** subcommands.	page 68
stop	Set a breakpoint for tasks in the current context. Breakpoints are stopping places in your program that halt execution.	page 65
trace	Set a tracepoint for tasks in the current context. Tracepoints are places in your program that, when reached during execution, cause the debugger to print information about the state of the program.	page 66
unhook	Unhook a task or group of tasks. Unhooking allows the task(s) to run without intervention from the debugger.	page 70
Note: In addition to the **pdbx** subcommands shown in this table, you can use most of the **dbx** subcommands. The **dbx** subcommands are all context sensitive. The only **dbx** subcommands that you cannot use are **clear**, **detach**, **edit**, **multproc**, **prompt**, **run**, **rerun**, and **screen**.		

Obtaining Information With pdbx Subcommands

This section describes three context insensitive **pdbx** subcommands you can use to obtain information. It describes how you can use the:

- **help** subcommand to display a list of **pdbx** subcommands.
- **dhelp** subcommand to display a list of **dbx** subcommands.
- **active** subcommand to display a list of active tasks.

Accessing Help for pdbx Subcommands

The **help** subcommand displays a list of **pdbx** subcommands. You can display this list as soon as you start the debugger. The partition does not yet need to be loaded. At the **pdbx** command prompt:

ENTER help

▲ The **pdbx** subcommands are listed

```
active                          - displays a list of active processes.
alias  [alias_name  string]     - sets or displays aliases.
delete <event_number>           - delete an event within the current context.
                                  The "event_number" can be found by
                                  issuing the "status" subcommand.
dbx    <dbx_subcommand>         - send a dbx subcommand directly to dbx.
dhelp                           - list synopsis of common dbx subcommands.
group  <action> [group_name] [member_list]
                                - used to manipulate groups. The actions
                                  are add, change, delete, and list.
                                  "member_list" is a list of tasks
                                  separated by spaces or commas.
help                            - "help" for pdbx subcommands.
hook                            - hooks a previously unhooked task.
load   <program> [program_options]
                                - load the program.
on     <context_specifier> [context_sensitive_subcommand]
                                - sets the current context for other
                                  subcommands, or executes a single
                                  subcommand using an explicit context.
quit                            - quits pdbx.
source <cmd_file>               - execute subcommands stored in a file.
status                          - displays the active trace and stop events
                                  within the current context.
stop   {variable | [variable] {at sourceline | in procedure}} [if condition]
                                - sets breakpoints within the current context.
trace  [sourceline | expression at sourceline | procedure |
       [variable] [at sourceline | in procedure] ] [if condition]
                                - sets tracepoints within the current context.
unalias                         - removes a previously created alias.
unhook                          - unhooks a task so it can run without pdbx
                                  intervention.
```

Accessing Help for dbx Subcommands

The **dhelp** subcommand displays a list of **dbx** subcommands. Unlike the **help** subcommand, the partition must be loaded before you can use this subcommand. At the **pdbx** command prompt:

ENTER **dhelp**

▲ A list of commonly used **dbx** subcommands appear. This list is the same as that displayed by entering **help** in regular **dbx**, so be aware that it includes some subcommands not supported by **pdbx**. Remember that you cannot use the **clear**, **detach**, **edit**, **multproc**, **prompt**, **run**, **rerun**, and **screen** subcommands in **pdbx**.

Displaying a List of Active Tasks

The **active** subcommand displays a list of active tasks. Active tasks are those tasks capable of receiving commands from the **pdbx** debugger. At the **pdbx** command prompt:

ENTER **active**

▲ The debugger displays a list similar to the following:

```
0    1    2    3    4
```

Setting Command Context and Grouping Tasks

This section describes how you can set and change the current command context. You can set the current command context on a specific task or group of tasks so that the debugger directs subsequent context sensitive subcommands to just that task or group. This section also shows how you can temporarily deviate from the current command context you set.

Setting the Current Command Context

When you begin a **pdbx** session, the default command context is set on all tasks. The **pdbx** command prompt always indicates the current command context setting, so it initially reads:

```
pdbx(all)
```

You can then use the **on** subcommand to set the current command context on one task or a subset of tasks. The debugger then directs context sensitive subcommands to just the task(s) specified by group or task name.

You can use the **on** subcommand to set the current command context *before* you load your partition. The debugger can only direct context sensitive subcommands to the tasks in the current context. The syntax of the **on** subcommand is:

on {*group_name* | *task_id*}

For example, assume you have a parallel program divided into tasks numbered 0 through 4. To set the current command context on just task 1:

ENTER **on** *1*

▲ The **pdbx** command prompt indicates the new setting of the current command context.

```
pdbx(1)
```

You can also use the **on** subcommand to set the current command context on all the tasks in a specified task group. The task group *all* – consisting of all tasks – is automatically defined for you. To set the command context back on all tasks, you would:

ENTER **on** *all*

▲ The **pdbx** command prompt shows that the current command context has changed, and that the debugger will now direct context sensitive subcommands to all tasks in the partition.

```
pdbx(all)
```

To deviate from the current command context for a single command, see "Temporarily Deviating From the Command Context" on page 62.

You can set the context on a group of tasks by first using the context insensitive **group** subcommand to collect a number of tasks under a common group name. None of these tasks need yet be active for you to include them in a group. Later, you can set the context on all the tasks in the group by specifying its group name with the **on** subcommand.

For example, you could use the **group** subcommand to collect a number of tasks (tasks 0, 1, and 2) as a group named *groupa*. Then, to set the context on tasks 0, 1, and 2, you would:

ENTER **on** *groupa*

▲ The debugger sets the command context on tasks 0, 1, and 2.

Another use of the **group** subcommand is to give a name to a task. For example, assume you have a typical master/slave program. Task 0 is the master task, controlling a number of slave tasks. You could create a group named *master* consisting of just task 0. Then, to set the context on the master task you would:

ENTER **on** *master*

▲ The debugger sets the command context on task 0. Entering **on** *master*, therefore, is the same as entering **on** *0*, but would be more meaningful and easier to remember.

The **group** subcommand has a number of actions. You can use it to:

- Create a task group, or add tasks to an existing task group
- Delete a task group, or delete tasks from an existing task group
- Change the name of an existing task group
- List the existing task groups, or list the members of a particular task group.

Adding a Task to a Task Group

To add a task to a new or already existing task group, use the **add** action of the **group** subcommand. The syntax is:

group add *group_name task_list*

If the specified *group_name* already exists, then the debugger adds the tasks in *task_list* to that group. If the specified *group_name* does not yet exist, the debugger creates it.

The variable *member_list* can be:	For example, to add the following task(s) to *groupa*:	You would ENTER:	▲ The following message displays:
a single task	task 6	**group add** *groupa 6*	`1 tasks added to group groupa`
a collection of tasks	tasks 6, 8, and 10	**group add** *groupa 6 8 10*	`3 tasks added to group groupa`
a range of tasks	tasks 6 through 10	**group add** *groupa 6,10*	`5 tasks added to group groupa`

Deleting Tasks from a Task Group

To delete tasks from a task group, use the **delete** action of the **group** subcommand. The syntax is:

group delete *group_name* **[***task_list***]**

The variable *member_list* can be:	For example, to delete the following from *groupa*:	You would ENTER:	▲ The following message displays:
a single task	task 6	**group delete** *groupa 6*	`6 deleted from group 'groupa'`

a collection of tasks	task 6, 8, and 10	**group delete** *groupa 6 8 10*	`6 deleted from group 'groupa'`
			`8 deleted from group 'groupa'`
			`10 deleted from group 'groupa'`

a range of tasks	tasks 6 through 10	**group delete** *groupa 6,10*	`6 deleted from group 'groupa'`
			`7 deleted from group 'groupa'`
			`8 deleted from group 'groupa'`
			`9 deleted from group 'groupa'`
			`10 deleted from group 'groupa'`

You can also use the **delete** action of the **group** subcommand to delete an entire task group. For example, to delete the task group *groupa*, you would:

ENTER **group delete** *groupa*

▲ The debugger deletes the task group.

Note: The pre-defined task group *all* cannot be deleted.

Changing the Name of a Task Group

To change the name of an existing task group, use the **change** action of the **group** subcommand. The syntax is:

group change *old_group_name new_group_name*

For example, say you want to change the name of task group *group1* to *groupa*. At the **pdbx** command prompt, you would:

ENTER **group change** *group1 groupa*

▲ The following message displays:

```
'groupa' is now the new name for 'group1'
```

Listing Task Groups

To list task groups and their members, use the **list** action of the **group** subcommand. The syntax is:

group list [*group_name*]

You can use the list subcommand to:	For example, if you ENTER:	▲

list all the task groups.	**group list**	The debugger displays a list of all existing task groups and their members. An example of such a list is shown below. Note that this lists only up to six members per task group.
		`all 0 A 1 U 2 U 3 U 4 U 5 U`
		`slaves 1 U 2 U 3 U 4 U 5 U`
		`master 0 A`
		`sgroup 3 U 5 U`

list all the members of a single task group	**group list** *groupa*	The debugger displays a list of all the members of task group *groupa*. `4 A 5 A 6 A 7 A 8 A 9 A 10 A`

When you list tasks, a single letter might follow each task number – either `A`, `E`, and `U`. The following table explains the meaning of these letters. If a task number is not followed by one of these letters, it means the task is not active.

This letter displayed after a task number:	Indicates that:
A	the task is active.
E	an error has occurred in communicating with the task.
U	the task is unhooked. Unhooking is a way of letting a task execute without debugger interaction. It is explained in greater detail in “Unhooking and Hooking Tasks” on page 70.

Temporarily Deviating From the Command Context

There are times when you need to deviate from the current command context for a single command. You can temporarily deviate from the command context by entering the **on** subcommand with, on the same line, a context sensitive subcommand. The syntax is:

on {*group_name* | *task_id*} [*subcommand*]

For example, assume a task group named *groupa* contains tasks 3 through 5. The current command context is on this group. You want to set a breakpoint at line 99 of task 3 only, and then continue directing commands to all three members of *group1*. One way to do this is to enter the three subcommands shown in the following example. This example shows the **pdbx** command prompt for additional illustration.

```
pdbx(groupa) on 3
pdbx(3) stop at 99
pdbx(3) on groupa
pdbx(groupa)
```

It is easier, however, to temporarily deviate from the current command context.

```
pdbx(groupa) on 3 stop at 99
pdbx(groupa)
```

The context sensitive **stop** subcommand is directed to task 3 only, but the current command context is unchanged. The next command entered at the **pdbx** command prompt is directed to all the tasks in the *groupa* task group.

Creating, Removing, and Listing Command Aliases

The **alias** subcommand specifies a command alias. You could use it to reduce the amount of typing needed, or to create a name more easily remembered. The syntax of this context insensitive subcommand is:

alias [*alias_name* **[***alias_string***]]**

For example, assume that you have organized all tasks into two convenient groups – *master* and *slaves*. During the execution of a program, you need to switch the command context back and forth between these two groups. You could save yourself some typing by creating one alias for *on slaves* and one for *on master*. At the **pdbx** command prompt, you would:

ENTER **alias** *mas on master*
alias *sla on slaves*

Now to turn the command context on the task group *master*, all you have to do is:

ENTER **mas**

Likewise, you can now enter **sla** instead of **on** *slaves*.

In addition to any aliases you create, there are a number of aliases supplied by **pdbx** when the partition is loaded. To display the list of all existing aliases, use the **alias** subcommand with no parameters. At the **pdbx** command prompt:

ENTER **alias**

▲ The debugger displays a list of existing aliases. The example listing below shows all the default aliases provided by **pdbx**, as well as the two aliases – *mas* and *sla* – created in the previous example.

```
t                   where
j                   status
st                  step
x                   registers
q                   quit
p                   print
n                   next
m                   map
l                   list
h                   help
d                   delete
c                   cont
mas                 on master
sla                 on slaves
```

Any aliases you create are not saved between **pdbx** sessions. You can also remove command aliases using the **unalias** subcommand. The syntax of this context insensitive subcommand is:

unalias *alias_name*

For example, to remove the alias *mas* defined above, you would:

ENTER **unalias** *mas*

Note: You can create, remove, and list command aliases as soon as you start the debugger. The partition does not need to be loaded.

Loading the Partition With the Load Subcommand

Before you can debug a parallel program with the **pdbx** debugger, you need to load your partition. If you specified a program name on the **pdbx** command, it is already loaded on each node of your partition. If not, you need to load your partition using the **load** subcommand. The Partition Manager allocates the nodes of your partition when you enter the **pdbx** command. It does this either by connecting to the Resource Manager or by looking to your host list file. The number of nodes in the partition depends on the value of the **MP_PROCS** environment variable (or the value specified on the **-procs** flag) when you enter the **pdbx** command.

How you load your partition depends on whether the program(s) you are debugging follow the SPMD or MPMD model of parallel programming. In the SPMD model, the same program runs on each of the nodes in your partition. In the MPMD model, separate programs can run on the nodes of your partition.

To load the same executable all nodes of the partition:	**To load separate executables on the partition:**
CHECK the **pdbx** command prompt to make sure the command context is on all tasks. The context *all* is the default when you start the **pdbx** debugger, so the prompt should read: `pdbx(all)` If the command context is not set on *all* tasks, reset it. To do this: **ENTER** **on all** Once the command context is on all tasks: **ENTER** **load** *program* [program_options] ▲ The specified program is loaded onto all nodes in the partition, and the message "Partition loaded..." displays. The parallel program runs up to the first executable statement and stops.	Set the command context before loading each program. For example, say your partition consists of five nodes numbered 0 through 4. To load a program named *program1* on node 0 and a program named *program2* on nodes 1 through 4, you would: **ENTER** **on** *0* ▲ The debugger sets the command context on node 0 **ENTER** **load** *program1* [program_options] The debugger loads *program1* on node 0. **ENTER** **group add** *groupa 1,4* ▲ The debugger creates a task group named *groupa* consisting of nodes 1 through 4. **ENTER** **on** *groupa* The debugger sets the command context on nodes 1 through 4. **ENTER** **load** *program2* [program_options] ▲ The debugger loads *program2* onto nodes 1 through 4, and the message "Partition loaded..." displays. The parallel program runs up to the first executable statement and stops.

Using Breakpoints and Tracepoints in pdbx

Like the **dbx** debugger, **pdbx** lets you set breakpoints and tracepoints to control and monitor program execution. *Breakpoints* are stopping places in your program. They halt execution, enabling you to then examine the state of the program. *Tracepoints* are places in the program that, when reached during execution, cause the debugger to print information about the state of the program. An occurrence of either a breakpoint or a tracepoint is called an *event*.

If you are already familiar with breakpoints and tracepoints as they are used in **dbx**, be aware that they work somewhat differently in **pdbx**. The subcommands for setting, checking, and deleting them are similar to their counterparts in **dbx**, but have been modified for use on parallel programs. These differences stem from the fact that they can now be directed to any number of parallel tasks.

This section describes how to:

- Set a breakpoint for tasks in the current context using the **stop** subcommand.
- Set a tracepoint for tasks in the current context using the **trace** subcommand.
- Use the **status** subcommand to display events set for tasks in the current context.
- Use the **delete** subcommand to remove events for tasks in the current context.

If you are already familiar with the **dbx** subcommands **stop**, **trace**, **status**, and **delete**, read the following as a discussion of how these subcommands are changed for **pdbx**.

If you are not familiar with **dbx**, read the following as an introduction to breakpoints and tracepoints. For more detail on the subcommands discussed, you can then refer to *AIX for RISC System/6000: Commands Reference* and *AIX for RISC System/6000: General Programming Concepts*.

Setting Breakpoints

The **stop** subcommand sets breakpoints for all tasks in the current context. When the tasks reach a breakpoint, execution stops and you can then examine the state of the program using other **pdbx** or **dbx** subcommands.

The syntax of this context sensitive subcommand is:

stop if *condition*

stop [*variable*] **at** *sourceline* [**if** *condition*]

stop [*variable*] **in** *procedure* [**if** *condition*]

stop *variable* [**if** *condition*]

Using the **stop** subcommand enables you to halt execution when:

- a specified *variable* is changed.
- a specified *sourceline* is reached using the **at** flag.
- a specified *procedure* is entered using the **in** flag.
- a specified *condition* is true using the **if** flag.

For example, to set a breakpoint at line 19 for all tasks in the current context, you would:

ENTER **stop at** *19*

▲ The debugger displays a message reporting the event it has built. The message includes the current context, the event ID associated with your breakpoint, and an interpretation of your command. For example:

```
all:[0] stop at "ftoc.c":19
```

The message reports that a breakpoint was set for the tasks in the task group *all*, and that the event ID associated with the breakpoint is *0*. Notice that the syntax of the interpretation is not exactly the same as the command entered.

Notes:

1. The **pdbx** debugger will not set a breakpoint at a line number when in a group context if the group members have different current source files. Instead, the following error message will be displayed.

   ```
   ERROR: Cannot set event in different source files.
   ```

   If this happens, you can either:

   - change the current context so that the **stop** subcommand will be directed to tasks with identical source files.
   - set the same source file for all members of the group using the **file** subcommand.

2. When specifying a variable name on the **stop** subcommand in **pdbx**, it is important to use fully-qualified names as arguments. See "Specifying Variables On the Trace and Stop subcommands" on page 67 for more information.
3. For further detail on the **stop** subcommand, refer to its use on the **dbx** command as described in *AIX for RISC System/6000: Commands Reference*.

Setting Tracepoints

The **trace** subcommand sets tracepoints for all tasks in the current context. When the tasks reach a tracepoint, they cause the debugger to print information about the state of the program.

The syntax of this context sensitive subcommand is:

trace

trace [*sourceline* | *expression* **at** *sourceline*] [**if** *condition*]

trace *procedure* [**in** *procedure*] [**if** *condition*]

trace [*variable*] [**in** *procedure* | **at** *sourceline*] [**if** *condition*]

The **trace** subcommand prints tracing information for a specified *procedure*, *function*, *sourceline*, *expression*, *variable*, or *condition*. For example, to set a tracepoint for the variable *foo* at line 21 for all tasks in the current context, you would:

ENTER **trace** *foo* **at** *21*

▲ The debugger displays a message reporting the event it has built. The message includes the current context, the event ID associated with your tracepoint, and an interpretation of your command. For example:

```
all:[1] trace foo at "bar.c":21
```

This message reports that the tracepoint was set for the tasks in the task group *all*, and that the event ID associated with the tracepoint is *1*. Notice that the syntax of the interpretation is not exactly the same as the command entered.

Notes:

1. The **pdbx** debugger will not set a tracepoint at a line number when in a group context if the group members have different current source files. Instead, the following error message will be displayed.

   ```
   ERROR: Cannot set event in different source files.
   ```

   If this happens, you can either:

   - change the current context so that the **trace** subcommand will be directed to tasks with identical source files.
   - set the same source file for all members of the group using the **file** subcommand.

2. When specifying a variable name on the **trace** subcommand in **pdbx**, it is important to use fully-qualified names as arguments. See "Specifying Variables On the Trace and Stop subcommands" for more information.
3. For further detail on the **trace** subcommand, refer to its use on the **dbx** command as described in *AIX for RISC System/6000: Commands Reference*.

Specifying Variables On the Trace and Stop subcommands

When specifying a variable name as an argument on either the **stop** or **trace** subcommand, you should use fully-qualified names. This is because, when the **stop** or **trace** subcommand is issued, the tasks of your program could be in different functions, and the variable name may resolve differently depending on a task's position.

For example, consider the following SPMD code segment in *myfile.c*. It is running as two parallel tasks – task 0 and task 1. Task 0 is in *func1* at line 4, while task 1 is in *func2* at line 9.

```
1  int i;
2  func1()
3  {
4      i++;
5  }
6  func2()
7  {
8      int i;
9      i++;
10  }
```

To display the full qualification of a given variable, you use the **which** subcommand of the **dbx** command. For example, to display the full qualification of the variable *i* if the current context is *all*:

ENTER **which** *i*

▲ The **pdbx** debugger displays the full qualification of the variable specified.

```
@myfile.i          (from line 1 of previous example)
@myfile.func2.i      (from line 8 of previous example)
```

Because the tasks are at different lines, issuing the following **stop** command would set a different breakpoint for each task:

stop if (*i* == *5*)

The debugger would display a message reporting the event it has built.

```
all:[0] stop if (i == 5)
```

The *i* for task 0, however, would represent the global variable (*@myfile.i*) while the *i* for task 1 would represent the local variable *i* declared within *func2* (*@myfile.func2.i*). To specify the global variable *i* without ambiguity on the **stop** subcommand, you would:

ENTER **stop if** (*@myfile.i* **==** *5*)

▲ The debugger reports the event it has built.

```
all:[0] stop if (@myfile.i == 5)
```

Checking Event Status

A list of **pdbx** events can be displayed by using the **status** subcommand. The syntax of this context sensitive subcommand is:

status

Each line of status displayed will have the following format:

```
context: [event-ID] event-description
```

The event ID displayed can later be used by the **delete** subcommand to remove the breakpoint or tracepoint.

Because the **status** command is context sensitive, it will not display status for events outside the context. Any events set for the task group *all*, however, will always be displayed by the **status** subcommand.

For example, say you have entered the following commands at the **pdbx** command prompt to set a number of breakpoints and tracepoints for the contexts shown.

on *all*
stop at *19*
trace *21*
on *0*
trace foo at 21
on *1*
stop in *func*

To display the events in the current context, you would:

ENTER **status**

▲ The debugger displays the list of events.

```
1:[0] stop in func
all:[0] stop at "foo.c":19
all:[1] trace "foo.c":21
```

Notice that status for task 0 is not displayed, since the context is now on task 1. Also notice that there is a unique event *0* for task 1 and for the task group *all*.

Deleting pdbx Events

The **delete** subcommand removes events of a specified event ID for the current context. The syntax of this context sensitive subcommand is:

delete *event_ID*

The event ID is the number associated with the breakpoint or tracepoint. This number is displayed by the **stop** and **trace** subcommands when an event is built. Event IDs can also be displayed using the **status** subcommand.

Event IDs are unique to the context in which they were set. Keep in mind that, in order to remove an event, the context must be on the appropriate task or task group. For example, say the current context is on task 1 and the output of the **status** subcommand is:

```
1:[0] stop in celsius
all:[0] stop at "foo.c":19
all:[1] trace "foo.c":21
```

To delete all these events, you would:

ENTER **on** *1*

delete *0*

on *all*

delete *0*

delete *1*

Controlling Execution With pdbx Subcommands

This section describes how to:

- Use the **source** subcommand to read **pdbx** subcommands from a specified commands file.
- Use the **unhook** subcommand so that tasks execute without interaction from the debugger.
- Interrupt process execution
- Use the **quit** subcommand to end a **pdbx** session.

Reading Subcommands From a Commands File

The **source** subcommand enables you to read a series of subcommands from a specified commands file. The syntax of this context-insensitive subcommand is:

source *commands_file*

The *commands_file* should reside on the Home Node, and can contain any of the subcommands that are valid on the **pdbx** command line. For example, say you have a commands file named *myalias* which contains a number of command alias settings. To read its commands:

ENTER **source** *myalias*

▲ The debugger reads the commands listed in *myalias* as if they had each been entered at the command line.

Note: You can also read commands from a file when starting the debugger. This is done using the **-c** flag on the **pdbx** command as described in Table 6 on page 55.

Unhooking and Hooking Tasks

The **unhook** subcommand lets you unhook a task so that it executes without intervention from the debugger. This subcommand is context sensitive and similar to the **detach** subcommand in **dbx**. The important difference is that you can regain control over a task that has been unhooked, while you cannot regain control over one that has been detached. To regain control over an unhooked task, use the **hook** subcommand.

To better understand the **hook** and **unhook** subcommands, consider the following example. You are debugging a typical master/slave program containing many blocking sends and receives. You have created two task groups. One – named *slaves* – contains all the slave tasks, and the other – named *master* – contains the master task. You would like to manipulate the master task and let the slave tasks process without debugger interaction. This would save you the bother of switching the command context back and forth between the two task groups. You could use the **detach** subcommand, but then you would never be able to reestablish control over the detached tasks. It would be better to unhook the *slaves* task group.

Since the **unhook** subcommand is context sensitive, you must first set the context on the *slaves* task group using the **on** subcommand. At the **pdbx** command prompt:

ENTER **on** *slaves*

▲ The debugger sets the command context on the task group *slaves*.

ENTER **unhook**

▲ The debugger unhooks the tasks in the task group *slaves*.

The slave tasks are still indirectly affected by the debugger since they might have to wait on a blocking receive for a message from the master task. However, they do execute without any direct interaction from the debugger. If you later wish to reestablish control over the tasks in the *slaves* task group, you would, assuming the context is on the *slaves* task group:

ENTER **hook**

▲ The debugger hooks any unhooked task in the current command context.

Interrupting Process Execution

When you enter a command at the **pdbx** command prompt, the prompt is not redisplayed until the process is complete. If the process cannot complete, or if it does not complete in a reasonable amount of time, you can interrupt it by pressing <**ctrl-c**>.

For example, assume you are debugging a parallel program containing many blocking sends and receives. The command context is set on one particular task, and you are using the **step** subcommand to move line by line through that task. Eventually you attempt to step past a blocking receive. The process cannot complete because the message the task is waiting for has not been sent. This is

because the task that normally sends this message has stopped at a breakpoint. The only way to get control back is to interrupt the process. To do this:

PRESS <**ctrl-c**>

▲ The **step** subcommand stops executing, and the **pdbx** command prompt appears.

Note: If the **pdbx** command prompt is not redisplayed when you press <**ctrl-c**>, then **pdbx** itself might be hung up. Pressing <**crtl-c**> twice terminates **pdbx** should this occur.

Stopping pdbx

Use the context insensitive **quit** subcommand to end a **pdbx** session. At the **pdbx** command prompt:

ENTER **quit**

Using the xpdbx Debugger

This section describes the **xpdbx** debugger. Like the **pdbx** debugger, **xpdbx** builds on the AIX debugger **dbx** by adding function specific to parallel programming. Unlike **pdbx**, the **xpdbx** debugger uses a Motif-based X-windows interface and does not require you to know the **dbx** subcommands. If you are familiar with **dbx** subcommands, however, you can enter them using the **xpdbx** Command Window as described in "Using the Command Window" on page 76.

Note: When certain displays or controls of the **xpdbx** debugger directly relate to specific subcommands in **dbx**, this section often points out that relation. This does not mean that knowledge of **dbx** or its subcommands is required to use the **xpdbx** debugger. If you are familiar with the **dbx** debugger, however, the details of how **xpdbx** relates to it should aid your understanding.

When you invoke a program using the **xpdbx** command, the **xpdbx** window automatically opens to mark the start of the debug session.

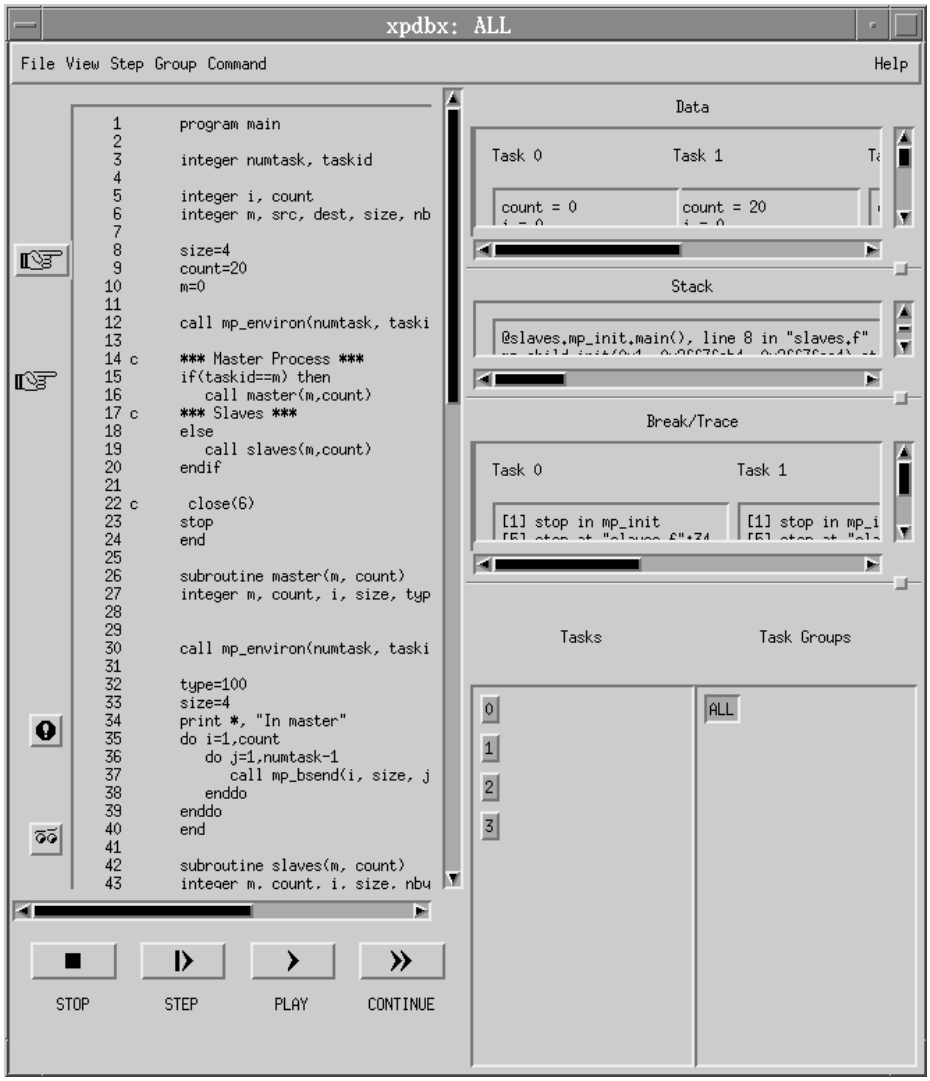


Figure 5. xpdbx window

This window consists of:

- A menu with the following options:
  - File
  - View
  - Step
  - Group
  - Command
  - Help
- The *Source Area*. This area displays the source code of the parallel program you are debugging. This area has both horizontal and vertical scroll bars for reading text displayed outside it.
- The *Data Area*. This area displays the values of the current routine's local variables. You can also select variables outside the current routine and display them in this area. In regular **dbx**, you would have to use a combination of the **dump** and **print** subcommands to display this information. The Data Area has both horizontal and vertical scroll bars for reading text displayed outside it.
- The *Stack Area*. This area displays the call stack for the current procedure or function. This is the same information you can display using the **where** subcommand in **dbx**. The Stack Area has both horizontal and vertical scroll bars for reading text displayed outside it.
- The *Break/Trace Area*. This area displays the active Break/Trace points for the tasks in the current context. This is the same information you can display using the **status** subcommand in **dbx**. The Break/Trace Area has both horizontal and vertical scroll bars for reading text displayed outside it.
- The *Basic xpdbx Controls*. These controls are directly below the Source Area and allow you to control the execution of the file you are debugging. These controls are similar to those you might find on a VCR or CD player, and are described in "Controlling Program Execution" on page 77.
- A *Task Area*. This area contains a number of *task* and *task group* push buttons that you can use to select tasks, or task groups, to be included in the current context. See "Setting the Context and Grouping Tasks" for more information.

Note: In this chapter, use the left mouse button for all menu selections and control operations unless otherwise noted.

Setting the Context and Grouping Tasks

Most of the controls and displays on the **xpdbx** window are sensitive to context. In **xpdbx**, *context* refers to a setting that specifies a certain task or group of tasks. The context is considered to be *on* the specified task(s), and the **xpdbx** window is considered to be *pointed at* the current context. While some of the displays and controls of the **xpdbx** window are insensitive to context, most of them only reflect or control the task(s) the window is currently pointed at.

When you start an **xpdbx** session, the context is set on all tasks. The title bar of the **xpdbx** window reads `xpdbx: ALL` to illustrate this. You can use the task and task group push buttons in the Task Area to set a new context. For example, assume you have a parallel program divided into five tasks. The tasks are numbered 0 through 4, and each has a task push button in the Task Area. To set the context on just task 1:

PRESS the task push button labeled *1* in the Task Area.

▲ The **xpdbx** debugger sets the context on task 1. To illustrate this, the debugger highlights the task's push button and updates the title bar of the **xpdbx** window to read `xpdbx: 1`.

You can also set the context on all the tasks in a particular task group. See "Creating Task Groups" for information on how to create task groups. The **xpdbx** debugger automatically defines the task group consisting of all tasks. This task group is named *ALL*. To set the command context back on the task group *ALL*:

PRESS the task group push button labeled *ALL* in the Task Area.

▲ The **xpdbx** debugger sets the context on all tasks. To illustrate this, the debugger highlights the **ALL** task group push button, all the task push buttons, and updates the title bar of the **xpdbx** window to read `xpdbx: ALL`.

Creating Task Groups

You can collect a number of tasks under a common group name. When you do this, the debugger creates a push button for the task group in the Task Area. You can then set the context on all the tasks in the group by pressing the push button. One task group – called *ALL* – is automatically defined by the debugger. It contains all active tasks.

To understand why you would want to define your own task groups, consider the following example. You are debugging a typical master/slave program containing many blocking sends and receives. The program has ten tasks. Task 0 is the master task, and tasks 1 through 9 are the slaves. During debugging you might start off by running the master until a blocking send or receive operation cannot complete. Then you could set the context on all the slaves and run them. Then set the context back on the master and run it some more.

Since you plan to keep switching the context back and forth between the master and slaves, you might find it helpful to group tasks 1 through 9 into a task group named *slaves*. Then you would only have to press a single task group button (instead of nine individual task buttons) to set the context on the slaves. If your program contains a greater number of slave tasks – say 63 – then grouping them becomes imperative.

You could also create a group named *master* containing just task 0. Although the "group" in this case has only one task, the name *master* is more meaningful than a task number and is therefore easier to remember.

To create a task group:

SELECT **Group → Create Group**

▲ The Add Group Dialog window opens.

FOCUS on the members area of the Add Group Dialog window.

TYPE IN task identifiers or names of already existing task groups to be included in the new group.

or

PRESS the task and task group push buttons of the tasks you wish to select.

FOCUS on the Group text entry field.

TYPE IN the name you want to give the group.

PRESS **OK**

▲ The Add Group Dialog window closes, and the debugger creates the new task group. The debugger creates a push button for the task group in the Task Area.

Deleting Task Groups

When a particular task group no longer seems necessary, you can delete it using the Delete Group Dialog window.

To delete a task group:

SELECT **Group → Delete Group**

▲ The Delete Group Dialog window opens.

PLACE the cursor over the name of the group.

PRESS the left mouse button.

▲ The group name highlights to show that you have selected it.

PRESS **OK**

▲ The debugger deletes the selected task group and closes the Delete Group Dialog window.

Loading the Partition

Before you can debug a parallel program with the **xpdbx** debugger, you need to load your partition. If you specified a program name on the **xpdbx** command, it is already loaded on each node of your partition. If not, you need to load your partition now. The Partition Manager allocates the nodes of your partition when you enter the **xpdbx** command. It does this either by connecting to the Resource Manager or by looking to your host list file. The number of nodes in the partition depends on the value of the **MP_PROCS** environment variable (or the value specified on the **-procs** flag) when you enter the **xpdbx** command.

How you load your partition depends on whether the program(s) you are debugging follow the SPMD or MPMD model of parallel programming. In the SPMD model, the same program runs on each of the nodes in your partition. In the MPMD model, separate programs are run on the nodes of your partition.

To load the same executable all nodes of the partition:	To load separate executables on the partition:
CHECK the title bar of the **xpdbx** window to make sure the context is on all tasks. It should read: `xpdbx: ALL` This is the default when you start the **xpdbx** debugger. If the context is not on the task group *ALL*, reset it. To set the context on all tasks: **PRESS** the task group button labeled *ALL* in the Task Area. Once the context is set on all tasks: **SELECT** **File → load** ▲ The Load Program Dialog window opens. This window contains a text entry field for the executable name. **FOCUS** on the Executable Name text entry field. **TYPE IN** the name of the program you want to load onto the partition. You can type program options into this field as well. Once you have typed in the program name and any options: **PRESS** **OK** ▲ The Program Dialog window closes, and the specified program is loaded onto all nodes in the partition. The parallel program runs up to the first executable statement and stops.	Set the context before loading each program. For example, say your partition consists of five nodes numbered 0 through 4. To load a program named *program1* on node 0 and a program named *program2* on nodes 1 through 4, you would: **PRESS** the task group button labeled *0* in the Task Area. **SELECT** **File → load** ▲ The Load Program Dialog window opens. This window contains a text entry field for the executable name. **FOCUS** on the Executable Name text entry field. **TYPE IN** the name of the program you want to load onto node 0. In this example, it is *program1*. If there are any program options, you can type them into this field as well. **PRESS** **OK** ▲ The Program Dialog window closes and the debugger loads *program1* on node 0. **PRESS** the task group buttons labeled *1*, *2*, *3*, and *4* in the Task Area. **SELECT** **File → load** ▲ The Load Program Dialog window opens. **FOCUS** on the Executable Name text entry field. **TYPE IN** the name of the program you want to load onto the rest of the partition. In this example, it is *program2*. If there are any program options, you can type them into this field as well. **PRESS** **OK** ▲ The Program Dialog window closes and the debugger loads *program2* onto nodes 1 through 4. The parallel program on each node runs up to the first executable statement and stops.

Using the Command Window

If you are familiar with the **dbx** debugger and its subcommands, you can use the Command window to issue **dbx** subcommands during an **xpdbx** session. The specified subcommand is issued to each task in the current context. To open the Command Window:

ENTER **Command → Command**

▲ The Command Window (shown in Figure 6 on page 77) opens.

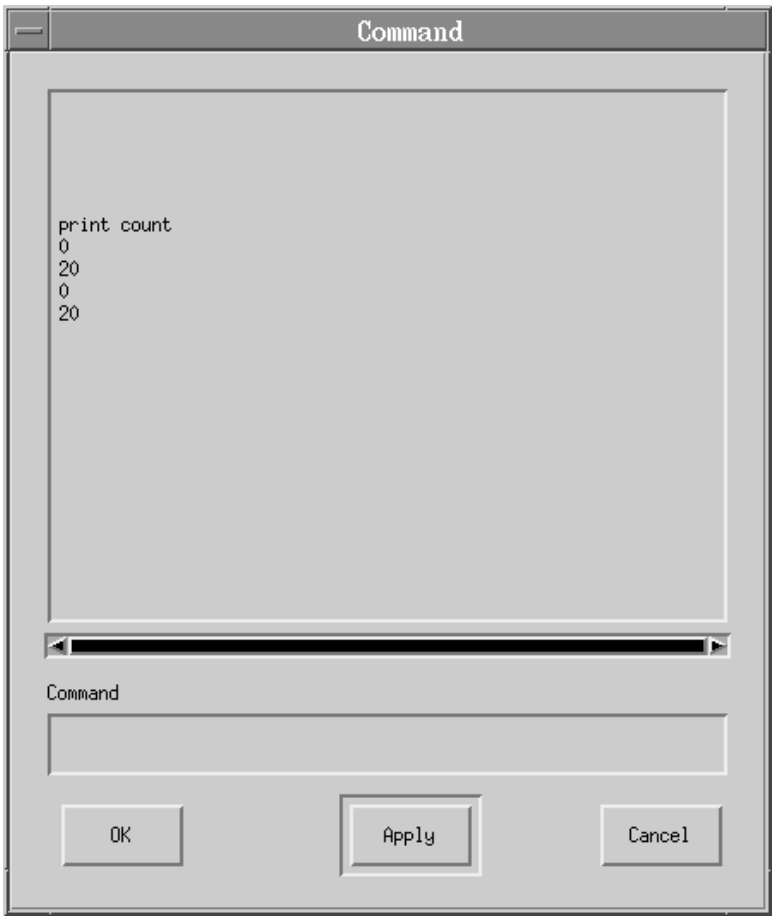


Figure 6. The Commands Window

To enter a **dbx** subcommand:

FOCUS on the Command text entry field.

TYPE IN the **dbx** subcommand you wish to enter.

PRESS **Apply**

or

OK

▲ The subcommand is issued to each task in the current context. **Apply** executes the command and displays any results in the Commands window. **OK** executes the command and closes the Commands window.

Controlling Program Execution

As with **dbx**, the **xpdbx** debugger lets you control execution by setting breakpoints in, or else by single stepping through, the source code. This section describes how to perform these familiar **dbx** tasks using the **xpdbx** debugger. It also describes some additional functions the **xpdbx** provides such as unhooking tasks so they can run without intervention from the debugger.

The basic method of controlling program execution with **xpdbx** is by manipulating the control buttons located directly below the Source Area on the **xpdbx** window. From left to right the control buttons are Stop, Step, Play, and Continue.

In order to:	Press this Control Button:
Stop execution of the parallel program for the tasks in the current context.	Stop
Manually step the execution of tasks in the current context by lines of source code.	Step
Have the debugger repeatedly step the tasks in the current context by lines of source code.	Play
Continue executing the tasks in the current context up to the next breakpoint or to the program's completion.	Continue

Every time execution of a program stops, the debugger updates the **xpdbx** window to display the current information. In the Source Area, the debugger uses a line marker to identify the line of code execution has stopped at. The debugger draws a line marker as a small hand pointing at the line of code. For example, when you first load a parallel program onto the partition, it runs up to the first executable statement and stops. In C programs, this is just before the main routine, so the debugger draws a line marker there. Since you can set the context on a subset of tasks and run them independent of the others, you can have a number of line markers displayed. When more than one task is at the same line of code, the line marker appears as a button. In Figure 5 on page 72, there are two line markers shown. One, representing a single task, is at line 15 of the program. The other, representing a group of tasks, appears as a button at line 8.

If you are unsure of the task(s) associated with a particular line marker:

PLACE the mouse cursor over the line marker.

PRESS the left mouse button.

▲ A label appears identifying the task(s) at that line of code. The label is visible only while you hold the mouse button down.

When more than one task is at the same line of code, you can manipulate them as a unit. This manipulation is independent of the current context. For example, say you are debugging a parallel program with three tasks numbered 0, 1, and 2. Tasks 0 and 1 are at line 22, while task two is at line 24. You want to move tasks 0 and 1 to the same line as task two. To move tasks 0 and 1 as a unit:

PLACE the mouse cursor over the line marker at line 22.

PRESS the right mouse button.

▲ A menu appears containing four items – Stop, Step, Play, and Continue. These menu items correspond to the four basic **xpdbx** control buttons.

SELECT **Step**

The debugger runs tasks 0 and 1 one line and stops them at line 23.

REPEAT these steps so that tasks 0, 1, and 2 are all at line 24.

Note: For more information on stepping, see “Stepping Execution” on page 81.

Setting Breakpoints

The **xpdbx** debugger lets you set stopping places, called breakpoints, in your program. You mark which lines are to be breakpoints for the tasks in the current context and then run the program. When the tasks reach a breakpoint, execution stops and you can then examine the state of the program. This is the same as using the **stop** subcommand in regular **dbx**.

You can set a breakpoint at a line of code or at the first executable statement in a procedure. This is the same as using the **at** or **in** options on the **stop** subcommand in regular **dbx**.

To set a breakpoint:

PLACE the mouse cursor over either a line number or a function/subroutine name in the Source Area.

DOUBLE-CLICK the left mouse button.

▲ The debugger sets, for the tasks in the current context, a breakpoint at the marked line of code or at the first executable statement in the marked function/subroutine. It is important to note that the debugger sets a separate breakpoint for each task in the context.

In the Source Area, the debugger places a stop marker (drawn to look like an stop sign containing an exclamation point) next to the line with the breakpoint. In Figure 5 on page 72, a stop marker appears at line 34 of the program.

In addition to the stop marker, the debugger displays a breakpoint event message (one for each of the tasks in the current context) in the Break/Trace area. The message includes an interpretation of the breakpoint preceded by the event ID associated with it. For example:

```
[1] stop at "ptst4.f":22
```

You can also specify a condition when setting a breakpoint. The tasks then stop executing at the breakpoint only if the condition evaluates to true. This is the same as using the **if** option on the **stop** subcommand in regular **dbx**. To set a conditional breakpoint:

PLACE the mouse cursor over either a line number or a function/subroutine name in the Source Area.

PRESS the left mouse button.

▲ The line number or function/subroutine name highlights to show that you have selected it.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Conditional Breakpoint**

▲ The debugger displays the Conditional Breakpoint Dialog window.

FOCUS on the text entry field of the Conditional Breakpoint Dialog window.

TYPE IN the condition that must evaluate to true for the execution to stop at the breakpoint. For example, to stop execution at the breakpoint only if the variable x is greater than 19, you would type in `x > 19`.

PRESS **OK**

▲ The debugger closes the Conditional Breakpoint Dialog window and sets a breakpoint for the tasks in the current context.

As with regular breakpoints, the debugger places a stop marker next to the line with the breakpoint in the Source Area. In the Break/Trace area, it adds a message reporting the conditional breakpoint the debugger has built for each of the tasks in the current context. For example:

```
[1] if x > 19 { stop } at "ptst4.f":22
```

Identifying the Tasks Associated with a Breakpoint: When you set a breakpoint following the previous instructions, it is important to remember that a separate breakpoint is set for each of the tasks in the current context. If you are unsure of the task(s) associated with a particular stop marker in the Source Area:

PLACE the mouse cursor over the stop marker.

PRESS the left mouse button.

▲ A label appears identifying the tasks with a breakpoint at that line of code. The label is visible only while you hold the mouse button down.

Removing Breakpoints: Any number of the active tasks may have a breakpoint at that same line of code. You can remove a breakpoint for:

- a single task
- all tasks with breakpoints at the same line of code.

The following table shows how to remove a breakpoint. This is the same as using the **delete** subcommand in regular **dbx**.

To remove the breakpoint for a single task:		**To remove all breakpoints at the same line of code:**	
PLACE	the mouse cursor over the breakpoint's event message for that task in the Break/Trace Area.	**PLACE**	the mouse cursor over the stop marker at that line of code.
	Note: The task must be in the current context for the event message to be displayed.	**PRESS**	the right mouse button.
PRESS	the left mouse button		▲ A selection menu appears.
	▲ The breakpoint's event message highlights to show that you have selected the breakpoint.	**SELECT**	**Delete**
PRESS	the right mouse button.		▲ The debugger removes all breakpoints set at that line of code. The stop marker disappears as well as the event messages displayed in the Break/Trace Area.
	▲ A selection menu appears.		
SELECT	**Delete**		
	▲ The breakpoint's event message disappears to show that the debugger has removed the breakpoint for the task.		

Stepping Execution

As with **dbx**, the **xpdbx** debugger lets you single-step execution of your program. In other words, you can run the tasks in the current context one source code line at a time. You can manually control each step, or have the debugger repeatedly step through the tasks in the current context. You can also specify whether you step into or over functions. By default, you step by lines of source code and over functions.

To manually step execution:

PRESS the Step Control Button.

▲ The debugger runs one line of the source code for the tasks in the current context and stops.

To repeatedly step execution:

PRESS the Play Control Button.

▲ The debugger repeatedly steps execution of the tasks in the current context by lines of the source code.

When repeatedly stepping tasks, execution continues for the tasks in the current context until the debugger reaches a breakpoint or you press the Stop Control Button. In this way, pressing the Play Control Button is similar to pressing the Continue Control Button. The important difference is that when repeatedly stepping execution, the debugger updates the **xpdbx** window after running each line of the source code. When continuing execution using the Continue Control Button, the debugger only updates the **xpdbx** window when execution reaches a breakpoint or you press the Stop Control Button.

Since the debugger has to keep updating the **xpdbx** window when repeatedly stepping tasks, this is the slower form of execution. The advantage is that it provides you with more information. For example, say you are debugging a typical master/slave program containing many blocking sends and receives. The context is on the slave tasks. You set a breakpoint and then press the Continue Control Button to continue execution of the slave tasks. Before reaching the breakpoint, however, the tasks hit a blocking receive intended to synchronize execution between the slaves and the master task. Because the master is not in the current context, the receive operation cannot complete and so the slaves cannot reach the breakpoint. Since the debugger does not update the **xpdbx** window until execution stops, the problem is not immediately observable. If you were repeatedly stepping the tasks, however, you would clearly see the line marker for the slaves stop just before the blocking receive.

Changing the Step Mode: When stepping execution, you can specify whether the debugger steps over functions or into them. This is called your *step mode*. Determining the step mode is the same as deciding whether to use the **step** or **next** subcommands in normal **dbx**.

For example, say you are stepping execution through a function that contains two calls to other functions located in separate source files. You know that the first function called does not cause errors, so you want to step over it. Stepping over functions is the default, so you simply step execution past the first function call. The debugger runs the lines in the called function but does not stop in it, and does not show you its source code. The second function called might have bugs, so you

want to step execution through it and closely examine its source code. To do this, you need to change the step mode to step into functions. You can change the step mode using the menu bar or the Step Control Button.

To change the step mode using the menu bar:	To change the step mode using the Step Control Button:
SELECT **Step → Step Into** ▲ The debugger changes the step mode, and toggles this menu item to read Step Over.	**PLACE** the mouse cursor over the Step Control Button. **PRESS** the right mouse button. ▲ A selection menu appears. **SELECT** **Step Into** ▲ The debugger changes the step mode, and toggles this menu item to read Step Over.

Now when you try to step execution past the function call, you step the tasks in the current context into the function. The debugger changes the source code displayed in the Source Area to that of the called function, and adds the function call to the Stack Area.

Customizing the Play Control Button: When you press the Play Control Button, it, by default, repeatedly executes steps at one-second intervals. However, you can customize the Play Control Button and specify:

- the command it should repeatedly execute
- the interval between the command iterations

To specify the command:	To specify the interval between command iterations:
PLACE the cursor over the Play Control Button. **PRESS** the right mouse button. ▲ A selection menu appears. **SELECT** **Change Command** ▲ The Change Command Dialog window opens. **FOCUS** on the text entry field in this window. **TYPE IN** the command you want the debugger to execute repeatedly when you press the Play Control Button. **PRESS** **OK** ▲ The Change Command Dialog window closes, and the Play Control Button is set to execute the command you specified.	**PLACE** the cursor over the Play Control Button. **PRESS** the right mouse button. ▲ A selection menu appears. **SELECT** **Change Interval** ▲ The Change Interval Dialog window opens. **FOCUS** on the text entry field in this window. **TYPE IN** The new interval length in seconds. **PRESS** **OK** ▲ The Change Interval Dialog window closes, and the Play Control Button is set to execute its command at the new interval you specified.

Examining Program Data

This section describes how to use the Data Area of the **xpdbx** Window to examine your program's data. This area shows the names and values of variables in the current routine. Each time execution of the program stops, the debugger automatically updates the information displayed. This is true whether the debugger stops because of a breakpoint, because you press the Stop Control Button, or because the program completes. In regular **dbx**, you would have to repeatedly issue the **dump** and **print** subcommands to get this information displayed.

The following table shows how variables are initially displayed in the Data Area.

In the Data Area, this type of variable:	Will initially be displayed:	For example:
scalar	with its value formatted according to its default type.	x = 300 a = -.0001 b = 331.789978 char_val = W
structure	with its type indicated.	struc_1 = MyOwnStruct_t structure
array	with its type indicated.	a = 10x2 array of int x = 5x5 array of float
pointer	with its type indicating	char_prt = pointer to char struc_ptr = pointer to structure
union	with its type indicated	MyOwnUnion_T = MyOwnUnion_T union

This section describes how, with appropriate mouse clicks in the Data Area, you can bring up menus and dialogs to:

- Display additional variables in the Data Area
- Display a variable in more or less detail
- Change a variable's value
- Enter values for the start and end range of an array index
- Change a variable's type
- Remove a variable from the Data Area

Displaying Additional Variables in the Data Area

The Data Area automatically displays the names and values of variables in the current routine. You can also select additional variables and display them in the Data Area.

To display an additional variable in the Data Area:

PLACE the cursor over the variable in the Source Area.

DOUBLE-CLICK the left mouse button.

▲ The debugger adds the variable to the Data Area.

Note: The amount of information the debugger can display for a selected variable depends on its scope. Global variables can always have their name and value displayed. If the current routine cannot reference the variable's value, however, the debugger only displays its name. For example:

x = unknown

Displaying More or Less Detail for a Variable

You can select a variable in the data area and display it in more or less detail. This will differ slightly depending on the type of variable you select. For example, you could select an array and then display its first element, or you could select a pointer and dereference it. You could also reverse this to display a variable in less detail.

To select a variable and display it in more or less detail:

PLACE the mouse cursor over a variable in the Data Area.

PRESS the left mouse button.

▲ The variable highlights to show that it is selected.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **More** or **Less**

▲ The variable displays in more or less detail. If the variable you selected cannot be displayed in more or less detail, it remains unchanged.

Changing a Variable's Value

You can select a variable in the data area and modify its value. You could modify the value of a variable to get it through a loop more quickly, or to test if the program is setting it to an incorrect value. You can also select an array index and change its range. This is particularly useful because, when you select an array as it is originally displayed, and display it in more detail as described in “Displaying More or Less Detail for a Variable” on page 83, the default range displayed is just one element. By changing the value of the array index you are able to study larger pieces of the array.

To select a variable and change its value:

PLACE the mouse cursor over a variable in the Data Area.

PRESS the left mouse button.

▲ The variable highlights to show that it is selected.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Change Value**

▲ The debugger displays the Change Value Dialog window.

FOCUS on the text entry field of the Change Value Dialog window.

TYPE IN the value you want to give the selected variable.

PRESS **OK**

▲ The debugger closes the Change Value Dialog window, and sets the variable to the value you specified.

Changing a Variable's Type

You can select a variable in the data area and modify its type. You could modify the type of a variable to:

- decimal
- letter
- hex
- octal
- bits
- string
- address

To select a variable and change its value:

PLACE the mouse cursor over a variable in the Data Area.

PRESS the left mouse button.

▲ The variable highlights to show that it is selected.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Change Type**

▲ A selection menu appears listing the possible types.

SELECT the type you want.

▲ The debugger formats the selected variable according to its new type.

Removing a Variable From the Data Area

Because the Data Area is a relatively small portion of the **xpdbx** window, variables can be difficult to read when many are displayed. For this reason, you may want to remove variables you are not interested in from this area. To remove a variable:

PLACE the cursor over the variable in the Data Area.

PRESS the left mouse button.

▲ The variable highlights to show that it is selected.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Hide**

▲ The debugger removes the selected variable from the Data Area.

Note: This option removes the entire variable. If all you selected was an array element or a structure field, the debugger removes the *entire* array or structure.

Hiding a Task's Data, Stack, and Break/Trace Information

On the **xpdbx** window, the Data, Stack, and Break/Trace areas present information for each task in the current context. There are times when you want to stop displaying information for a particular task or task group in one or more of these areas. The main reason you might want to do this would be to improve the debugger's performance. Another reason could be to conserve space in the area and improve the readability of information still displayed there.

For example, say you are debugging a master/slave program. The program has 63 identical slave tasks and you are stepping execution through them. Since the information displayed for each task is essentially the same, you might want to hide all but one. The information will then be easier to read and the screen refreshes will be faster.

To hide a task's data, stack, and break/trace information:

PLACE the mouse cursor over a task button or a task group button in the task area of the **xpdbx** window.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT either **Hide Data**, **Hide Stack**, **Hide Break/Trace**, or **Hide All**

▲ The debugger no longer displays information for the task in the specified window. If you selected **Hide All**, the debugger hides the tasks information in all three areas. When you hide a window, the **Hide** option on the selection menu toggles to **Show**. You can then repeat these steps to again display the task's information in the specified window.

Tracing Program Execution

The **xpdbx** debugger lets you set tracepoints in your program. When tasks reach a tracepoint during execution, they write information regarding the state of the program to the Command window. See "Using the Command Window" on page 76 for more information on this window.

You can set a tracepoint at a line of code or at the first executable statement in a procedure. This is the same as using the **at** or **in** options on the **trace** subcommand in regular **dbx**.

To set a tracepoint:

PLACE the mouse cursor over either a line number or a function/subroutine name in the Source Area.

PRESS the left mouse button.

▲ A selection menu appears.

SELECT **Tracepoint**

▲ The debugger sets, for the tasks in the current context, a tracepoint at the marked line of code or at the first executable statement in the marked function/subroutine. It is important to note that the debugger sets a separate tracepoint for each task in the context.

In the Source Area, the debugger places a trace marker next to the line with the tracepoint. The trace marker is drawn as two eyes looking at the line of code. In Figure 5 on page 72, a trace marker appears at line 40 of the displayed source code.

In addition to the trace marker, the debugger displays a tracepoint event message (one for each of the tasks in the current context) in the Break/Trace area. The message includes an interpretation of the tracepoint preceded by the event ID associated with it. For example:

```
[6] trace at "mikia.f":15
```

You can also specify a condition when setting a tracepoint. The tasks then write trace information only if the condition evaluates to true. This is the same as using the **if** option on the **trace** subcommand in regular **dbx**. To set a conditional tracepoint:

PLACE the mouse cursor over either a line number or a function/subroutine name in the Source Area.

PRESS the left mouse button.

▲ The line number or function/subroutine name highlights to show that you have selected it.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Conditional Tracepoint**

▲ The debugger displays the Tracepoint Dialog window.

FOCUS on the text entry field of the Tracepoint Dialog window.

TYPE IN the condition that must evaluate to true for trace information to be written. For example, `x > 19`.

PRESS **OK**

▲ The debugger closes the Tracepoint Dialog window and sets a tracepoint for the tasks in the current context.

As with regular tracepoints, the debugger places a trace marker next to the line with the tracepoint in the Source Area. In the Break/Trace area, it adds a message reporting the conditional tracepoint for each of the tasks in the current context. For example:

```
[7] if @blist.mp_init.main.i > 4 {   } at "blist.f":23
```

Identifying the Tasks Associated with a Tracepoint: When you set a tracepoint following the previous instructions, it is important to remember that a separate tracepoint is set for each of the tasks in the current context. If you are unsure of the task(s) associated with a particular trace marker in the Source Area:

PLACE the mouse cursor over the trace marker.

PRESS the left mouse button.

▲ A label appears identifying the tasks with a tracepoint at that line of code. The label is visible only while you hold the mouse button down.

Removing Tracepoints: Any number of the active tasks may have a tracepoint at that same line of code. You can remove a tracepoint for:

- a single task
- all tasks with tracepoints at the same line of code.

The following table shows how to remove a tracepoint. This is the same as using the **delete** subcommand in regular **dbx**.

To remove the tracepoint for a single task:		To remove all tracepoints at the same line of code:	
PLACE	the mouse cursor over the tracepoint's event message for that task in the Break/Trace Area.	**PLACE**	the mouse cursor over the trace marker at that line of code.
	Note: The task must be in the current context for the event message to be displayed.	**PRESS**	the right mouse button. ▲ A selection menu appears.
PRESS	the left mouse button ▲ The tracepoint's event message highlights to show that you have selected the tracepoint.	**SELECT**	**Delete** ▲ The debugger removes all tracepoints set at that line of code. The trace marker disappears as well as the event messages displayed in the Break/Trace Area.
PRESS	the right mouse button. ▲ A selection menu appears.		
SELECT	**Delete** ▲ The tracepoint's event message disappears to show that the debugger has removed the tracepoint for the task.		

Changing the Displayed Source Code

During an **xpdbx** session, the source code file displayed in the Source Area may change many times. Each time execution of the program stops, the debugger updates the **xpdbx** window and checks that the source code displayed is current. If execution has stopped, for example, in a procedure located in a different file, the debugger automatically updates the Source Area to display the current source. You can also change the current source displayed by:

- Selecting a function or subroutine from the Source or Stack Area.
- Opening a source code file using the File Selection Dialog.

Displaying a Function's Source Code

You can update the Source Area to display the source code of a function or subroutine you select. To do this:

PLACE the mouse cursor over the name of a function or a subroutine in either the Source Area or the Stack Area.

PRESS the left mouse button.

▲ The function/subroutine name highlights to show that you have selected it.

PRESS the right mouse button.

A selection menu appears.

SELECT **Show Source**

▲ The debugger changes the source code displayed in the Source Area to that of the selected function/subroutine.

Opening a Source Code File

You can open a source code file and display it in the Source Area using the File Selection Dialog window. To do this:

SELECT **File → Open**

▲ The File Selection Dialog window opens.

This window contains:

- A filter area showing the current search path
- a list of directories in the current search path
- a list of files in the current search path.

If the source code file is not in your current search path, you can specify a new search path. To do this:

FOCUS on the filter area.

TYPE IN the new search path.

PRESS **Filter**

▲ The **xpdbx** debugger updates the lists of directories and files accordingly.

To select a source code file:

PLACE the cursor over the entry in the list of files.

PRESS the left mouse button.

▲ The full path name of the source code file appears in the selection field.

PRESS **OK**

▲ The File Selection Dialog window closes, and the source code of the selected file appears in the Source Area.

Debugging Programs Using Multiple Views

You can think of the **xpdbx** window as a *view* into the *debug space* The view you have is just one way of looking into the debug space, and depends on the current context, the source code displayed in the Source Area, and so on. You can open multiple **xpdbx** windows, however, and so have multiple views into the same debug space.

For example, say you have two tasks – tasks 0 and 1 – involved in message passing. You could open two **xpdbx** windows to follow send and receive pairs between the two tasks. In one window, you would set the context on task 0. In the other, you would set the context on task 1. You could then step execution past the send in one window, and then step execution past the receive in the other.

When dealing with multiple views into the same debug space, keep in mind that actions made on one **xpdbx** window may be reflected on the others. For example, say you have two views into the same debug space. The context of one is set on just task 0, while the context of the other is set on all the tasks including task 0. If you step all the tasks in the second window, the first window also reflects the step.

To open another **xpdbx** window to provide an additional view into the debug space:

SELECT **View → Open new view**

▲ Another **xpdbx** window opens.

To close a **xpdbx** window:

SELECT **View → Close this view**

Chapter 4. Profiling Parallel Programs

This chapter describes how to use the AIX profilers **prof** and **gprof** on parallel programs. It is intended for application developers already familiar with **prof** and **gprof**, and so does not describe how to use these commands to identify problem areas in, or tune, parallel programs. For information on the use of **prof** and **gprof**, consult *AIX for RISC System 6000: Commands Reference*.

The difference between profiling serial and parallel applications with the AIX profilers is that serial applications can be run to generate a single profile data file, while a parallel application can be run to produce many.

Following the traditional method of profiling using the AIX operating system, you compile a serial application and run it to produce a single profile data file that you can then process using either the **prof** or **gprof** commands. With a parallel application, you compile and run it to produce a profile data file for each parallel task. You can then process one, some, or all the data files produced using either the **prof** or **gprof** commands. The following table describes how to profile parallel programs. For comparison, the steps involved in profiling a serial program are shown in the left-hand column of the table.

To Profile a Serial Program:	**To Profile a Parallel Program:**
Step 1: Compile the application source code using the **cc** command with either the **-p** or **-pg** flag.	Step 1: Compile the application source code using the command **mpcc** (for C programs), **mpCC** (for C++ programs), or **mpxlf** (for FORTRAN programs) as described on page 8. You should use one of the standard profiling compiler options – either **-p** or **-pg** – on the compiler command. For more information on the compiler options **-p** and **-pg**, refer to their use on the **cc** command as described in *AIX for RISC System/6000: Commands Reference* and *AIX for RISC System/6000: General Programming Concepts*.
Step 2: Run the executable program to produce a profile data file. If you have compiled the source code with the **-p** option, the data file produced is named *mon.out*. If you have compiled the source code with the **-pg** option, the data file produced is named *gmon.out*.	Step 2: Run the parallel program. As the program runs, it generates a profile data file for each parallel task. The system gives unique names to the data files by appending each task's identifying number to *mon.out* or *gmon.out*. If you have compiled the source code with the **-p** option, the data files produced take the form: **mon.out.***taskid* If the source code has been compiled with the **-pg** option, the data files produced take the form: **gmon.out.***taskid* **Notes:** 1. The current directory must be writable from all remote nodes. Otherwise, the profile data files will have to be manually moved to the home node for analysis with **prof** and **gprof**. 2. To obtain subroutine call frequency counts, you must add */usr/lib/profiled* to your **LIBPATH** environment variable before running your program.

© Copyright IBM Corp. 1993

To Profile a Serial Program:	To Profile a Parallel Program:
Step 3: Use either the **prof** or the **gprof** command to process the profile data file. You use the **prof** command to process the *mon.out* data file, and the **gprof** command to process the *gmon.out* data file.	Step 3: Use either the **prof** or **gprof** command to process the profile data files. The **prof** command processes the *mon.out* data files, and **gprof** processes the *gmon.out* data files. You can process one, some, or all of the data files created during the run. You must specify the name(s) of the profile data file(s) to read, however, because the **prof** and **gprof** commands read *mon.out* or *gmon.out* by default. On the **prof** command, use the **-m** flag to specify the name(s) of the profile data file(s) it should read. For example, to specify the profile data file for task 0 with the **prof** command: **ENTER** **prof -m mon.out.***0* You can also specify that the **prof** command should take profile data from some or all of the profile data files produced. For example, to specify three different profile data files – the ones associated with tasks 0, 1, and 2 – on the **prof** command: **ENTER** **prof -m mon.out.***0* **mon.out.***1* **mon.out.***2* On the **gprof** command, you simply specify the name(s) of the profile data file(s) it should read on the command line. You must also specify the name of the program on the **gprof** command, but no option flag is needed. For example, to specify the profile data file for task 0 with the **gprof** command: **ENTER** **gprof** *program* **gmon.out**.*0* As with the **prof** command, you can also specify that the **gprof** command should take profile data from some or all of the profile data files produced. For example, to specify three different profile data files – the ones associated with tasks 0, 1, and 2 – on the **gprof** command: **ENTER** **gprof** *program* **gmon.out.***0* **gmon.out.***1* **gmon.out.***2*

Chapter 5. Visualizing Program and System Performance

This chapter describes the Parallel Environment's Visualization Tool (VT). VT is a group of displays, or *views*, which show unique performance characteristics of an application program and your system. Each view presents specific, often complex, information in some familiar and easily-interpretable form. For example, a view could be a bar chart, a strip graph, a pie chart, or a grid. Since there are many different views, you open the ones most appropriate to your needs and the information you wish to see visualized.

You can use the VT views for trace visualization and online performance monitoring.

- In *trace visualization*, you play back statistical and event records – or *trace records* – generated during a program's execution. You can use VT to visualize information about the program as well as its use of the underlying system. This visualized information can help you tune the program – optimize its use of the underlying system. "Using VT for Trace Visualization" on page 105 describes how to run a program to generate files containing trace records, and how to play back these trace records using VT.

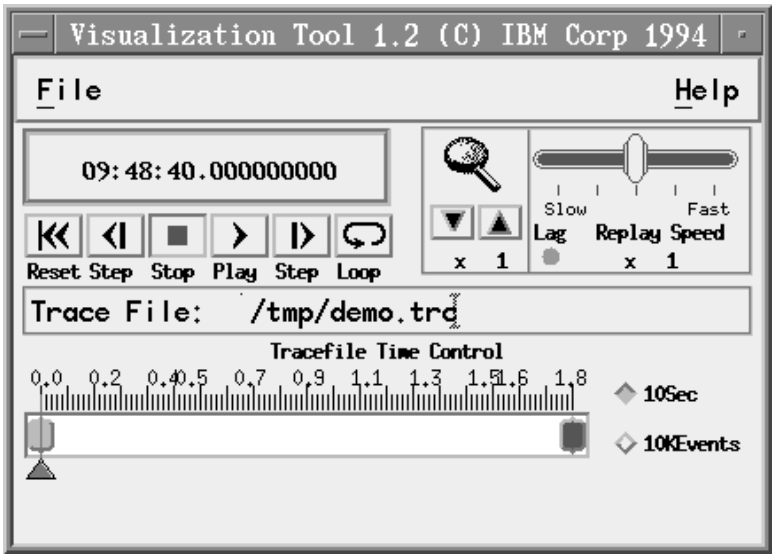


Figure 7. VT Window for Trace Visualization

- In *performance monitoring*, you use VT as an online monitor to study the operational status and activity of each of the processor nodes in your POWERparallel system or RISC System/6000 network cluster. In performance monitoring, VT only displays system statistics and not communication information. See "Using VT for Performance Monitoring" on page 120.

© Copyright IBM Corp. 1993

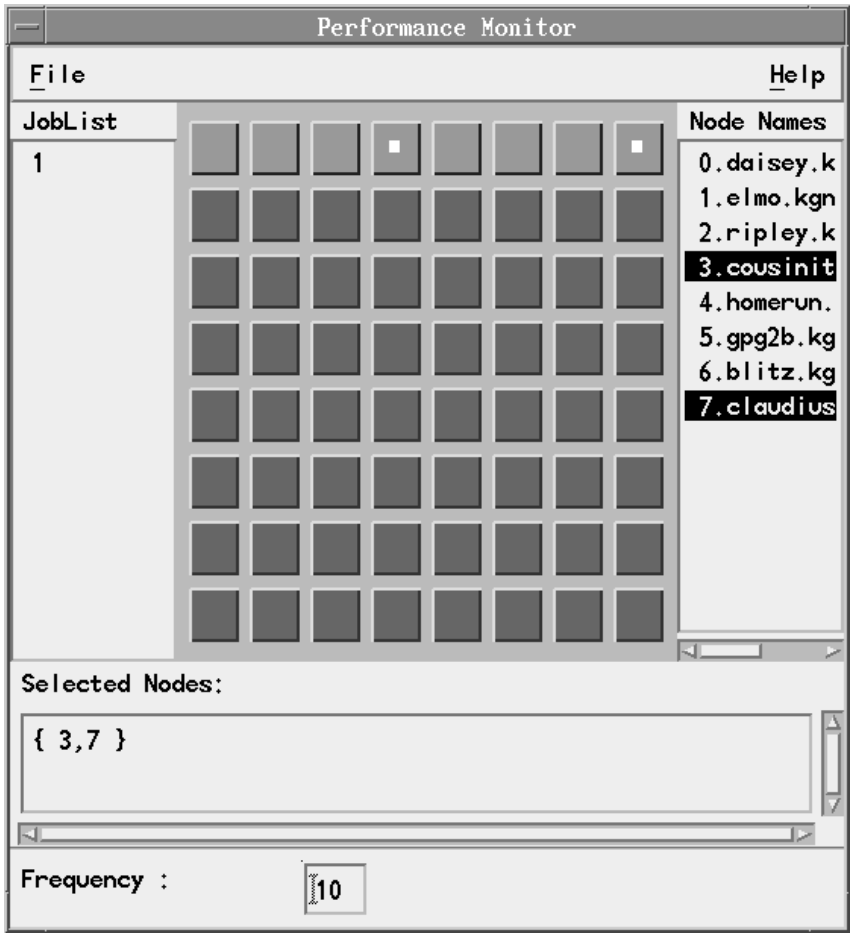


Figure 8. VT Window for Performance Monitoring

The VT views enable you to visualize:

- communication among processor nodes
- the type and duration of communication events
- CPU utilization of processor nodes
- a parallel program's source code as it relates to the executable's run
- the number of times a program's tasks are swapped in and out of active execution on a processor node
- the number of disk reads, disk writes, and disk transfers for a processor node
- the number of TCP/IP packets sent and received by a processor node.
- the number of times a processor node has to load a page of virtual memory back into real memory
- the number of times a processor node invokes a kernel subroutine
- a summary of system activity.

Often, a number of views take the same information and present it in different ways. For example, one might use a bar chart, and another a strip graph. This allows you to select not only the information you wish to visualize, but also the form. “Opening and Closing Views” on page 98 contains a general description of

views and their use. “View Descriptions” on page 124 contains a more detailed description of the views, the information each presents, and how to interpret them.

Table 7 on page 96 is designed for those who wish to start using VT immediately for either trace visualization or performance monitoring. While the remainder of this chapter takes a slower, more detailed, approach to describing the steps involved in generating and playing a trace file and monitoring system activity, this table does not. It is designed to get you started as quickly as possible, and so little detail or explanation is given for each step. Many readers may prefer the more detailed approach to VT, which begins with “Starting the Visualization Tool” on page 97.

Note: For additional information of VT, refer to Appendix D, “Visualization Tool Hints and Tips for Application Developers” on page 227. This appendix is a separate document designed to supplement the information included in this chapter.

Table 7. VT Quick Operation

For Trace Visualization:

- Step 1: Compile your program using the command **mpcc** (for C programs), **mpCC** (for C++ programs), or **mpxlf** (for FORTRAN programs). You must use the standard **-g** compiler option to generate an object file needed by the VT Source Code view. For more information, see page 107.
- Step 2: Invoke the parallel program using the POE command-line flag **-tlevel**.

  ENTER **poe** *program* **-tlevel** *9*

  ▲ The program executes, generating a trace file called *program.trc*.
- Step 3: After *program* has finished executing, start a VT session and begin playback of the trace file. If you have not generated a trace file of your own, use *vtsample.trc* (as shown in the instruction below) for demonstration.

  ENTER **vt -tfile** */usr/lpp/vt/tracefiles/vtsample.trc*

  ▲ The Trace Visualization window and the View Selector window open. If you get an error saying that the message catalog is not found, set the **NLSPATH** environment variable to **/usr/lib/nls/msg/%L/%N** and the **LANG** environment variable to **En_US** and start VT again.
- Step 4: Open views to visualize the trace records. To do this:

  PRESS the following icons in the View Selector window:
  - Source Code
  - User Load Balance
  - Interprocessor Communication

  ▲ The three views open. You can select any of the views and are not limited to these. To interpret the information these views present, see "View Descriptions" on page 124.
- Step 5: Start playback of the trace file. To do this:

  PRESS the Play Control Button in the Trace Visualization window.

  ▲ VT begins playing back the trace records stored in *vtsample.trc*. The trace records are visualized in the three open views.
- Step 6: Stop Playback of the trace file. To do this:

  PRESS the Stop Control Button on the Trace Visualization window.

  ▲ VT stops playing back *vtsample.trc*.
- Step 7: End the VT session. To do this:

  SELECT **File → Exit**

For Performance Monitoring:

- Step 1: Start a VT session for performance monitoring.

  If you are in an environment where the POWERparallel system Resource Manager is available:

  ENTER **vt**

  If you are in an environment where the POWERparallel system Resource Manager is not available:

  ENTER **vt -norm**

  ▲ The Trace Visualization window and the View Selector window open. If you get an error saying that the message catalog is not found, set the **NLSPATH** environment variable to **/usr/lib/nls/msg/%L/%N** and the **LANG** environment variable to **En_US** and start VT again.

  SELECT **File → Performance Monitor**

  ▲ The Performance Monitor window and the PM View Selector window open. Each of the squares on this window represents a processor node of your POWERparallel system or cluster. The processor nodes are also listed by name in the Node Name List, and, if you are using a POWERparallel system, each of the jobs running are listed in the Jobs List.

  Note: Before you start monitoring, you should first select nodes and open views.
- Step 2: Select processor nodes for monitoring.

  PLACE the cursor over one of the squares on this window, over the name of a processor node in the Node List, or over the name of a job in the Job List.

  PRESS the left mouse button.

  ▲ If you placed the cursor over one of the squares on the window, or over the name of a processor node in the Node List, that processor node is selected for monitoring. If you placed the cursor over the name of a job in the Job List, then all the processor nodes that job is running on are selected for monitoring.
- Step 3: Open views to visualize the kernel statistics being sent from the selected processor nodes. In the PM View Selector window:

  PRESS the Computation category push button.

  ▲ All the Computation views open. To interpret these views, see "View Descriptions" on page 124.
- Step 4: Start monitoring the selected processor nodes.

  SELECT **File → Monitor**

  ▲ Each of the selected processor nodes starts sending samplings of AIX kernel statistics to VT.
- Step 5: End the monitoring session. To do this:

  SELECT **File → Done**

Starting the Visualization Tool

The following table describes how to start VT for either trace visualization or performance monitoring. You can also start VT from the Parallel Desktop by double-clicking on its icon.

Note: For National Language Support, VT displays messages located in an externalized message catalog. 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=En_US

The PE message catalog is in English, and is located in *usr/lib/nls/msg/En_US*. If your site is using its own translation of the message catalog, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For additional information on NLS and message catalogs, see *IBM AIX Parallel Environment: Installation, Administration, and Diagnosis* and *AIX for RISC System/6000: General Programming Concepts*.

Table 8. Starting VT

To start VT for trace visualization:	**To start VT for performance monitoring if you are in an environment where the POWERparallel system Resource Manager is available:**	**To start VT for performance monitoring if you are in an environment where the POWERparallel system Resource Manager is not available:**
ENTER **vt** ▲ The Trace Visualization window and the View Selector window automatically open, marking the start of a VT session. After reading the following description of the **vt** command-line flags and the next section on how to open and use views, go to "Using VT for Trace Visualization" on page 105.	**ENTER** **vt** ▲ The Trace Visualization window and the View Selector window automatically open, marking the start of a VT session. **SELECT** **File → Performance Monitor** ▲ The Performance Monitor window and the PM View Selector window open. After reading the following description of **vt** command-line flags and the next section on how to open and use views, go to "Using VT for Performance Monitoring" on page 120.	**ENTER** **vt -norm** ▲ The Trace Visualization window and the View Selector window automatically open, marking the start of a VT session. **SELECT** **File → Performance Monitor** ▲ The Performance Monitor window and the PM View Selector window open. After reading the following description of **vt** command-line flags and the next section on how to open and use views, go to "Using VT for Performance Monitoring" on page 120.

There are also a number of optional command-line flags you can use on the **vt** command. These are summarized in the following table:

Table 9. VT Command-line Flags

Use this command-line flag:	To:	For example:	For more information, see:
-tracefile or **-tfile**	Automatically load a specified trace file for playback when starting VT.	**vt -tracefile** *tracefile* or **vt -tfile** *tracefile*	page 115
-go	Start playing back the trace file immediately upon starting VT. When you use this flag, you must also specify a tracefile and a configuration file using the **-tracefile** (or **-tfile**) flag and the **-configfile** (or **-cfile**) flags.	**vt -tfile** *tracefile* **-cfile** *config* **-go**	page 116
-configfile or **-cfile**	Load a configuration file. These files contain previously saved arrangements of VT windows, as well as input field specifications.	**vt -configfile** *config* or **vt -cfile** *config*	page 104
-cmap	Request a private color map. When you use this flag, VT makes a private copy of the default X-Windows color map.	**vt -cmap**	page 102
-norm	Indicate that you are not using the Resource Manager. If you are using a RISC System/6000 network cluster, you must use this option.	**vt -norm**	table 8
-spath	Indicate a search path to a program's source code. Like the AIX **PATH** environment variable, this is a series of colon-delimited directory names to search. Unless the program's source is in the current directory, the search path is needed to display it in the Source Code view. You can also indicate a search path to a program's source code using the Source Code view.	**vt -spath** */u/files/source:/u/hink/source*	page 132

Opening and Closing Views

Whether you are using VT for trace visualization or performance monitoring, it provides a set of displays called views. You can open and close these views from the View Selector window. These views show, or *visualize*, specific information about your program or system in easily interpretable forms such as bar charts and strip graphs. Using mouse clicks, you can display details of the displayed information or change a view's appearance or configuration. Some VT views are for trace visualization only, and some are for both trace visualization and performance monitoring.

There are two types of views – *instantaneous* and *streaming*. Instantaneous views present information for a specific point in time, while streaming views represent a range of time. On a streaming view, a vertical line drawn towards the right of the display shows the current point of trace playback. If doing performance monitoring, this line represents the current point in time.

VT arranges the views into the following *view categories*:

- Communication/Program
- Computation
- Disk
- Network
- System

The view(s) in this category:	Visualize:
Communication/Program	information regarding message-passing events between processor nodes while running a particular program. Also shows the source code of a program whose trace records are being played back. These views are for trace visualization only.
Computation	information regarding the utilization of the processor nodes running a particular program. These views can be used for performance monitoring or trace visualization.
Disk	information regarding the number of disk reads, disk writes, and disk transfers. These views can be used for trace visualization or performance monitoring.
Network	information regarding the number of TCP/IP packets sent or received by processor nodes. These views can be used for trace visualization or performance monitoring.
System	information regarding system activities and events such as page faults and context switches. These views can be used for performance monitoring or trace visualization.

Whether you are using VT for trace visualization or performance monitoring, you want to open the views most appropriate to the events or statistics you wish to examine.

If you are doing trace visualization, for example, and are interested in studying the message passing events that occurred between processor nodes as your program ran, you would open the Interprocessor Communication and Message Status Matrix views. If you want to see the actual lines of source code associated with the message passing events, you would also open the Source Code view.

You can open and close these views as you see fit during a VT session. For example, as you play a particular trace file, you might initially be interested only in the processor load balance as shown in the User Load Balance view. If the load balance is particularly skewed, however, you might want to then look at some system views.

Note: Views in the Communication/Program category should not be opened once playback of a tracefile has begun, but this restriction does not apply to views in other categories.

You can open and close views by selecting their icons in the View Selector window. The icons in this window are labeled by view name and grouped by view category. Each view category has a labeled push button.

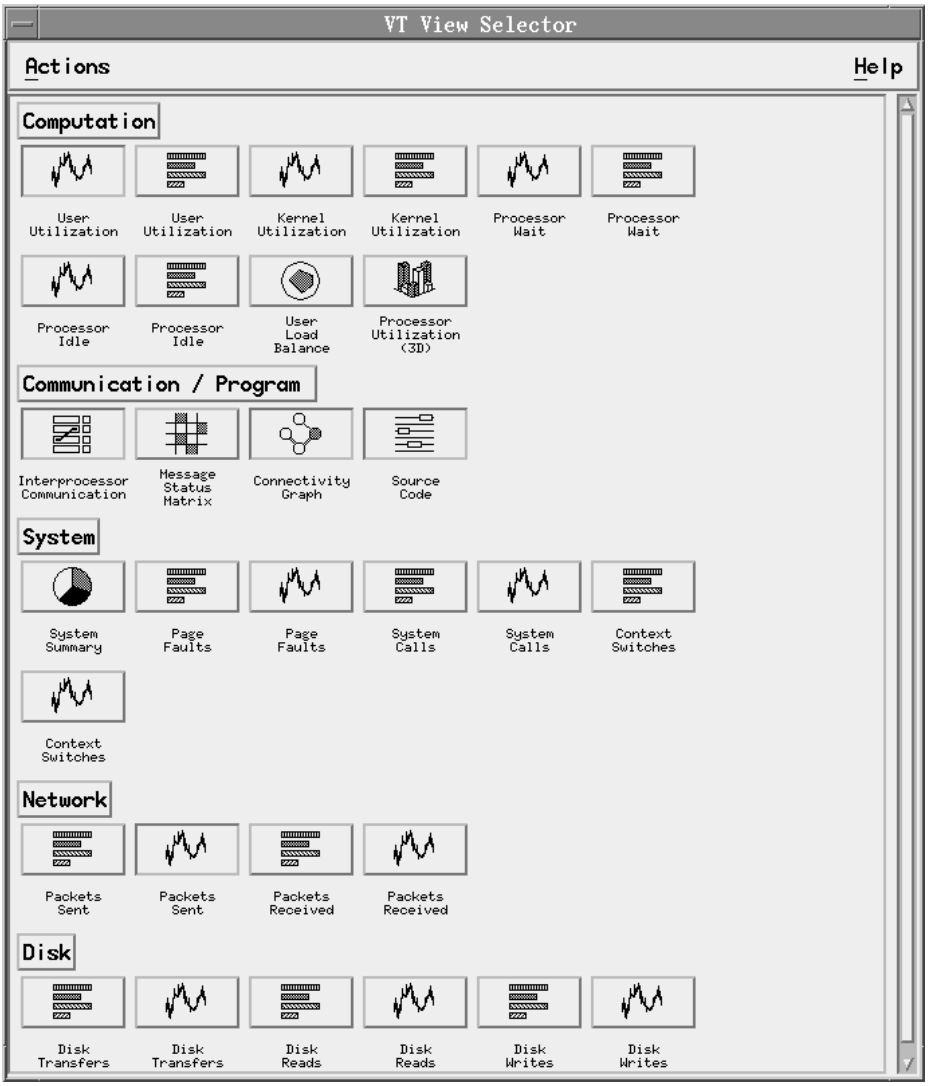


Figure 9. View Selector Window

To select a single view from the View Selector window.

PRESS the icon associated with the view in the View Selector window.

▲ The view opens. If already open, it closes.

To select all the views in a particular view category from the View Selector window.

PRESS the labeled push button for that view category.

▲ If there are any unopened views in the category, they are opened. Otherwise, all the views in the category are closed.

Note: "View Descriptions" on page 124 contains a description of each of the views in the five view categories. Refer to this section to decide on the view(s) most appropriate for your purposes.

Adjusting a View's Display Rate and Colors

When doing trace visualization or performance monitoring, views redisplay using new information from VT after set intervals. By default, this interval is 0.1 second and is called the *display rate*. The actual appearance of the view – the colors it uses – is determined by its *display spectrum*. You can change the display rate and display spectrum for each view. To adjust a view's display rate and select from a fixed set of display spectrums:

PLACE the cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **configuration**

▲ The view's Time Resolution window opens.

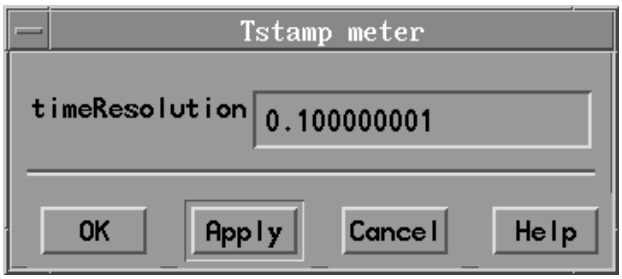


Figure 10. Time Resolution Window

FOCUS on the Time Resolution input field.

TYPE IN the new display rate in seconds.

PRESS **OK**

▲ The view's Parameters window opens. This window lets you change the appearance of a view. All views let you change the display spectrum. Some views allow you to set other parameters as discussed in “View Descriptions” on page 124.

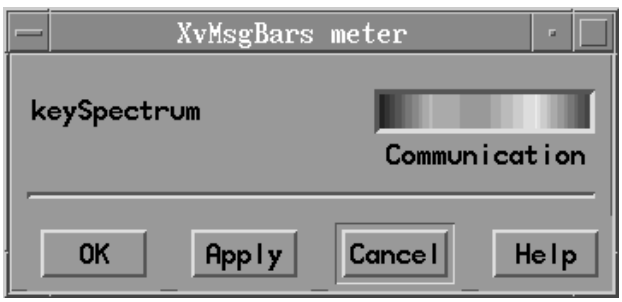


Figure 11. Parameters Window

The Parameters window lets you change the appearance of the view. To change to a new display spectrum:

PLACE the cursor over the key spectrum field.

PRESS the left mouse button.

▲ A different display spectrum appears in the key spectrum field. The name of the display spectrum appears below the field. You can continue clicking on this field to see each display spectrum for the view. You can also use the middle mouse button to display the available spectrums in reverse order.

PRESS OK

▲ The Time Resolution window and Parameters window close. The view will now redisplay itself at its new set rate using its new display spectrum.

Notes:

1. To open a view's Parameters window directly, bypassing the Time Resolution window:

   PLACE the cursor over the view.

   PRESS the right mouse button.

   ▲ A selection menu appears.

   SELECT Parameters

2. Many of the views allow you to adjust more than just their display rate and color. “View Descriptions” on page 124 describes, for each of the views, any additional parameters you can adjust.

3. VT needs a large number of colors for the various display spectrums used by the views. By default, VT attempts to use the default color map shared by all active X-Windows applications. Depending on the number of active X-Windows applications, there might not be enough available colors for VT. When this happens, VT displays a message indicating the spectrum(s) it cannot allocate, and uses black in place of the unallocated color(s). VT will still run, but in extreme cases some display spectrums may be unusable because of the missing color(s).

   One way to avoid this problem is to have fewer competing X-Windows applications when you start VT. You can also instruct VT to create its own private copy of the color map using the **-cmap** flag on the **vt** command when starting VT. To do this:

   ENTER vt -cmap

   If you have another X-Windows application running and VT is using its own color map, window colors may change as you move focus between applications. When VT is active, the colors in the other windows may change. When VT is not active, its colors may change.

Saving and Loading a VT Configuration File

A configuration file controls the windows that are displayed, where they appear on your screen, and what values they show. When you have a screen configuration you like, you can save it in a configuration file. Later you could load that configuration file to display the same screen configuration.

For example, say you are doing trace visualization of a program's run and have five or six views open. Based on the information you see visualized, you plan to modify the program so it runs more efficiently. When you have made the changes, you will rerun the program to generate a new trace file and then play it back using VT. You know you will want to use these same five or six views, so you decide to save the screen configuration. To do this:

SELECT **File → Save Configuration**

▲ The Save Configuration dialog window opens.

FOCUS on the text entry field of the dialog window.

TYPE IN the name you want to give the configuration file.

PRESS **OK**

▲ The screen configuration is saved as the file name you specified.

To later load this saved configuration:

SELECT **File → Load Configuration**

▲ The Configuration File Selection Dialog window opens.

This window contains:

- A filter area showing the current search path
- A list of the directories in the current search path
- A list of the files in the current search path.

If the configuration file you want to load is not in the current search path, you can specify a new search path. To do this:

FOCUS on the filter area.

TYPE IN the new search path.

PRESS **Filter**

▲ VT updates the list of directories and files accordingly.

To load a configuration file:

PLACE the cursor over the configuration file name in the list of files.

PRESS the mouse button.

▲ The full path name of the configuration file appears in the Selection field.

PRESS **OK**

▲ VT loads the configuration file and closes the Configuration File Selection Dialog window.

Note: You can also load a configuration file when starting a VT session. To do this, use the **-cfile** or **-configfile** flag on the **vt** command. For example, say you want to load the configuration file *myconfig* located in the directory */home/brady/john*. You would:

ENTER **vt -configfile** */home/brady/john/myconfig*

or

vt -cfile */home/brady/john/myconfig*

▲ The VT session starts, using the screen configuration saved in the configuration file */home/brady/john/myconfig*.

Using VT for Trace Visualization

Trace visualization enables you to play back statistical and event records generated during a program's execution. These statistical and event records are called *trace records* and are stored in a *trace file*. By playing a trace file and opening views to visualize its trace records, you can perform algorithm characterization and study your program's performance. This section discusses:

- The four types of trace records
- How to generate a trace file
- How to play back the trace file and visualize its trace records using VT.

Types of Trace Records

There are four types of trace records. They are:

- Message Passing
- Collective Communication
- AIX Kernel Statistics
- Application Markers

The following sections describe each type of trace record in more detail. In addition, the header file *VT_trc.h* in the directory */usr/include* defines the structure of trace records. Refer to this header file if you want to read, manipulate, or interpret trace records independent of VT. Refer also to */usr/lpp/vt/README.traces*, which describes the VT tracing architecture.

Message Passing

Message passing trace records contain information regarding simple message-passing events such as blocking sends and receives among tasks of your program. Each of these events is the result of a call to an MPL subroutine. MPL subroutines are described in greater detail in *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference*.

Collective Communication

Collective communication trace records contain information about communication events involving groups of tasks. Broadcasts and combines are examples of collective communication trace records. Each of these events is the result of a call to an MPL subroutine. MPL subroutines are described in greater detail in *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference*.

AIX Kernel Statistic

AIX kernel statistic trace records contain a sampling of statistics from the kernel. These include the:

- Percent of CPU utilization (user, kernel, wait, and idle)
- Number of system calls
- Number of page faults
- Number of transfers to and from disk
- Number of blocks read from disk
- Number of blocks written to disk
- Number of TCP/IP packets received
- Number of TCP/IP packets sent

Application Marker

This type of trace record contains marker information created for application calls. You may code markers using the Parallel Utility Function mpc_marker (for C programs) or MP_MARKER (for FORTRAN programs). When you run a program, you can display these markers online using the Program Marker Array as described in "Monitoring Program Execution Using the Program Marker Array" on page 46. When you later play back a trace file of the program's run using VT, the marker information can again be displayed – this time in the Source Code view. See the description of the Source Code view on page 131. The mpc_marker and MP_MARKER Parallel Utility Function calls are described in *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference*.

Generating Trace Files

To generate trace files:

1. You have the option to turn trace generation off and on within your application program. You can select which type(s) of trace records you wish to switch off or on. This ability (described in "Step 1: Control Trace Record Generation Within the Program") enables you to generate trace files containing just the types of trace records (for just the part of the program) that you are interested in visualizing. In order for a trace record type to be generated, however, it must also be enabled by the **MP_TRACELEVEL** environment variable, or its associated command-line flags when you invoke your program. See "Step 3: Process the Program to Generate a Trace File" on page 108 for more information.
2. Compile your program as described in "Step 2: Compile the Program" on page 107.
3. Invoke the compiled program with trace record generation turned on. You can turn trace record generation on for one, some, or all trace record types. See "Step 3: Process the Program to Generate a Trace File" on page 108.

Step 1: Control Trace Record Generation Within the Program

VT uses its own routines to create trace records and does not utilize the AIX trace facility. Some of these can be called from your application program, allowing you to generate trace files containing just the type(s) of trace records you are interested in visualizing. To turn tracing off or on, use the following prototypes:

For FORTRAN programs	For C programs
`INTEGER FUNCTION VT_TRC_STOP` `INTEGER FUNCTION VT_TRC_START ( INTEGER FLAG )`	`int VT_trc_stop_c()` `int VT_trc_start_c( int )`

The variable *int_flag* is an integer flag that specifies one, some, or all trace record types. The following table shows the possible values of *int_flag* and the trace record type(s) indicated by each.

Table 10. Trace Record Integer Flags				
Flag Value	**Message Passing**	**Collective Communication**	**AIX Kernel Statistic**	**Application Markers**
0				
1				√
2			√	√
3	√	√		√
9	√	√	√	√

Let us say you wanted trace records to be generated for only the second half of a program, and that you felt only AIX Kernel Statistics and Application Markers trace records were necessary for your purposes.

1. At the top of the program, you would add the following line.

For FORTRAN Programs	For C Programs
`VT_trc_stop()`	`VT_trc_stop_c()`

   This stops the generation of all trace record types for the first half of the program.

2. In the second half of the program, to start generating Message Passing and Collective Communication trace records, you would then insert the following line at the appropriate place in your program.

For FORTRAN Programs	For C Programs
`VT_trc_start(2)`	`VT_trc_start_c(2)`

Notes:

1. Turning tracing off and on from your program may result in misleading information being displayed in the Interprocessor Communication view. For more information, see the note on page 126.
2. During normal trace file playback, VT attempts to simulate actual time. If you have turned trace record generation off for, say, five minutes of a program's run, VT does not automatically skip over that area of the trace file. During normal playback, you will have to wait the five minutes before any of the views are updated. You can get around this by advancing playback over the next trace record as described in “Step 5: Stepping Playback” on page 117.

Step 2: Compile the Program

In order to take advantage of all VT features, you need to use the **-g** option as shown below when compiling your program with the **mpcc**, **mpCC**, or **mpxlf** command. These compiler commands are actually shell scripts that call the regular **cc**, **xlC**, or **xlf** compiler. The **-g** option produces an object file with symbol table references needed to take advantage of the Source Code view. This view lets you see the actual lines of code associated with the trace record events you are visualizing. The **-g** flag is not required if you do not wish to use the Source Code view.

Notes:

1. For more information on the **mpcc**, **mpCC**, and **mpxlf** commands, see page 8 and Appendix A, “Parallel Environment Commands” on page 165.
2. For more information on the **-g** option, refer to its use on the **cc** command as described in *AIX for RISC System/6000: Commands Reference*.

Step 3: Process the Program to Generate a Trace File

To generate a trace file, you need to process your program with tracing turned on. You can turn tracing on by setting the environment variable **MP_TRACELEVEL**, or using the **-tracelevel** or **-tlevel** flag when invoking the program. By default the trace level is 0, meaning that tracing is off. As with all POE command-line flags, **-tracelevel** and **-tlevel** override their associated environment variable.

Whether you set the **MP_TRACELEVEL** environment variable or use one of the command-line flags, you use an integer to indicate the trace record type(s) you wish to generate. As the program runs, it generates a trace file in the current directory. By default, trace file names match the program name with the suffix *.trc* added.

The integer you use with the **MP_TRACELEVEL** environment variable and its associated flags are the same ones you use on the *VT_trc_start* statement within your programs, and are detailed in Table 10 on page 107. For example, to generate a trace file for all trace record types you could:

Set the MP_TRACELEVEL environment variable:	**Use the -tracelevel or -tlevel flag when invoking the program:**
ENTER **export MP_TRACELEVEL=***9*	**ENTER** **poe** *program* **-tracelevel** *9* or **poe** *program* **-tlevel** *9*

Notes:

1. When you process with tracing on for trace record types, your program can turn tracing off and on for the type(s) specified using the *VT_trc_stop* and *VT_trc_start* routines. When you process without turning tracing on for trace record types, the program has no control over tracing for the type(s) specified.
2. If you are generating a trace file with AIX Kernel Statistic trace records, you should carefully consider whether or not you want to time share processor nodes with the other tasks of your program or with other users. If you run one task per processor node and do not share the CPU with other users, you will have a record of each task's use of the underlying system. If you time share nodes, you still have a record of each processor node's kernel statistics. These statistics, however, do not help you analyze your program's use of the system.
3. VT generates some useful informational messages during trace generation. In order to get these messages, the **MP_INFOLEVEL** environment variable, or its associated flag **-infolevel**, must be set to level *2* or higher. Refer to “Setting the Message Reporting Level for Standard Error (STDERR)” on page 37 for more information on **MP_INFOLEVEL**.

Specifying a Trace File Name: By default, VT names trace files the same as the program name with the suffix *.trc* added. There are times when you do not want to use the default name and want to specify your own. To specify a trace file name, you can set the environment variable **MP_TRACEFILE** or use the **-tracefile** or **-tfile** flag when invoking the program. As with all POE command-line flags, **-tracefile** or **-tfile** temporarily override their associated environment variable.

For example, say you generate a trace file for *program*. You do not specify a name, so VT names it *program.trc* by default. You play back the trace records in *program.trc*, and based on the information visualized decide to modify *program* so it runs more efficiently. You make the modifications and now want to process *program* to generate a second trace file. You do not want to overlay *program.trc*, however, so you need to give this one a new name. To name the trace file *tracefile2.trc*, you could:

Set the MP_TRACEFILE environment variable:	**Use the -tracefile or -tfile flag when invoking the program:**
ENTER **export MP_TRACEFILE=***tracefile2*	**ENTER** **poe** *program* **-tlevel** *9* **-tracefile** *tracefile2* or **poe** *program* **-tlevel** *9* **-tfile** *tracefile2*

Changing the Sampling Interval for AIX Kernel Statistics: If you are generating trace records for AIX Kernel Statistics, VT gets a sampling of those statistics at set intervals. By default, the set interval is every twenty milliseconds. You can change the sampling interval by setting the environment variable **MP_SAMPLEFREQ**, or using the **-samplefreq** or **-sfreq** flag when invoking the program. As with all POE command-line flags, **-samplefreq** and **-sfreq** temporarily override their associated environment variable.

If you sample the AIX Kernel Statistics more frequently, your resulting trace file will be more detailed, but will also require more storage. If storage is a consideration, you might want to sample AIX Kernel Statistics less frequently. For example, say you want to get a sampling of AIX Kernel Statistics every 50 milliseconds. You could:

Set the MP_SAMPLEFREQ environment variable:	**Use the -samplefreq or -sfreq flag when invoking the program:**
ENTER **export MP_SAMPLEFREQ=***50*	**ENTER** **poe** *program* **-tlevel** *9* **-samplefreq** *50* or **poe** *program* **-tlevel** *9* **-sfreq** *50*

Note: You can also set the sampling interval from within your application program. To do this, use the routine VT_trc_set_params (for FORTRAN programs) or the routine VT_trc_set_params_c (for C programs). See *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference* for more information.

Managing Storage for Trace Files: By default, the system uses the following three-tiered approach to manage the storage of trace files. First, it writes the trace records to a 1 MB buffer in memory. Then the system flushes the buffers contents, when filled, to a temp file (by default */tmp/userid*) with 10 MB of memory. When the temp file would exceed 10 MB, the system flushes its contents to your current directory. When execution of the program completes, all the individual trace files

are then integrated into one final trace file. The default maximum size of the final trace file is 10 MB. Figure 12 on page 110 illustrates the three-tiered approach used to manage the storage of trace files.

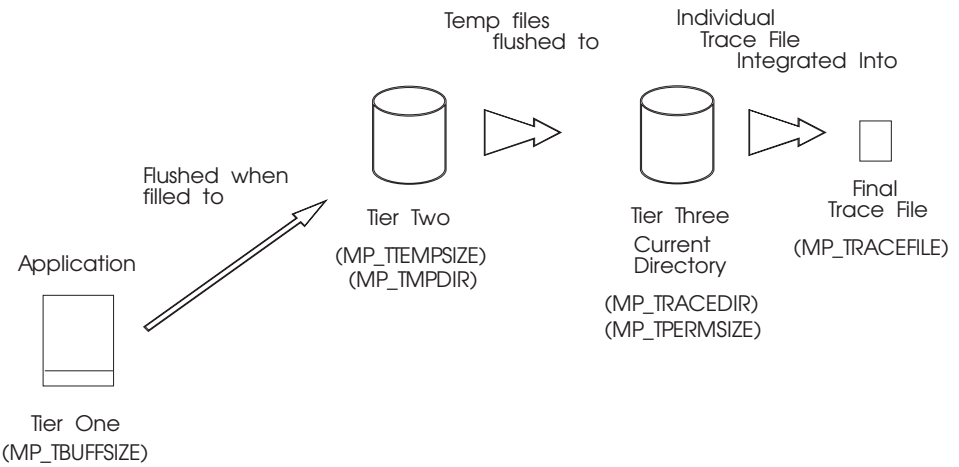


Figure 12. Three-Tiered Trace File Storage Approach

You have several options regarding the storage of trace files. You can specify:

- a file other than */tmp/userid* for the temporary trace file.
- a directory other than your current directory for final trace file output.
- the maximum size of the buffer, the temp file, and the permanent trace file.
- a wraparound storage approach instead of the default three-tiered approach. With this approach, the system overwrites the buffer instead of flushing it to a temp file.

Writing the Temporary Trace File to a Specified Directory: By default, VT writes the memory buffer to a file */tmp/userid*. You can have it write the temporary trace file to a different directory by setting the **MP_TMPDIR** environment variable or using the **-tmpdir** flag when invoking the program. If this directory is a local file system, it will minimize the network traffic. As with all POE command-line flags, the **-tmpdir** flag temporarily overrides its associate environment variable.

For example, to write a trace file to *tracefiles* in your home directory, you could:

Set the MP_TMPDIR environment variable:	**Use the -tmpdir flag when invoking the program:**
ENTER **export MP_TMPDIR=***$HOME/tracefiles*	**ENTER** **poe** *program* **-tlevel** *9* **-tmpdir** *$HOME/tracefiles*

Writing the Final Trace File to a Specified Directory: By default, VT writes the final trace file to the your current directory. Each time the temporary file fills, it is written to this directory. When execution of the program completes, all these individual trace files are then integrated into the final trace file. Although VT builds the final trace file in your current directory by default, you can use the environment variable **MP_TRACEDIR** to specify a different directory. You can also temporarily override the value of **MP_TRACEDIR** using one of its associated command-line flags – either **-tracedir** or **-tdir**. To specify that VT should build the final trace file in the directory */u/salat/cris*, you could:

Use the MP_TRACEDIR environment variable:	Use the -tracedir or -tdir flag when invoking the program:
ENTER **export MP_TRACEDIR=***/u/salat/cris*	**ENTER** **poe** *program* **-tlevel** *9* **-tracedir** */u/salat/cris* **poe** *program* **-tlevel** *9* **-tdir** */u/salat/cris*

Specifying the Buffer, Temp File, and Trace File Size: There are times when you will want to increase the size of the buffer, the temp file, or the trace file. For example, each time the system flushes the contents of your buffer or temp file, some of its resources are not available to run your program. By increasing the size of the buffer and temp file, you could decrease the number of times this happens. Also, the amount of trace files generated from a run can be quite large and may exceed the default 10 MB limit. For these reasons, you can specify:

- the size of the buffer by setting the **MP_TBUFFSIZE** environment variable or using the **-tbuffsize** or **-tbsize** command-line flag.
- the size of the temp file by setting the **MP_TTEMPSIZE** environment variable or using the **-ttempsize** or **-ttsize** command-line flags.
- the maximum size of the output file by setting the **MP_TPERMSIZE** environment variable or using the **-tpermsize** or **-tpsize** command-line flags.

Note: You can also manage storage for trace files from within your application program. To do this, use the routine VT_trc_set_params (for FORTRAN programs) or the routine VT_trc_set_params_c (for C programs). See *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference* for more information.

Say you wanted to increase the size of the buffer to 5 MB, the size of the temp file to 20 MB, and the maximum size of the trace file to 300 MB. To do this, you could:

Set the three environment variables:	Use the command-line flags:
ENTER **export MP_TBUFFSIZE=***5M* **export MP_TTEMPSIZE=***20M* **export MP_TPERMSIZE=***300M*	**ENTER** **poe** *program* **-tlevel** *9* **-tbuffsize** *5M* **-ttempsize** *20M* **-tpermsize** *300M* or **poe** *program* **-tlevel** *9* **-tbsize** *5M* **-ttsize** *20M* **-tpsize** *300M*

Specifying a Wraparound Storage Approach: You can change to a wraparound storage approach instead of the default three-tiered approach by setting the **MP_TBUFFWRAP** environment variable or using the **-tbuffwrap** or **-tbwrap** command-line flag when invoking a parallel program. With a wraparound storage approach, the system overwrites the buffer instead of flushing it to a temp file. As with all POE command-line flags, **-tbuffwrap** and **-tbwrap** temporarily override their associated environment variable. To specify a wraparound storage approach, you could:

Set the MP_TBUFFWRAP environment variable:	Use the -tbuffwrap or -tbwrap flag when invoking the program:
ENTER **export MP_TBUFFWRAP=***yes*	**ENTER** **poe** *program* **-tlevel** *9* **-tbuffwrap** *yes* or **poe** *program* **-tlevel** *9* **-tbwrap** *yes*

Using VT to Play Trace Files

The Trace Visualization window automatically opens at the start of a VT session.

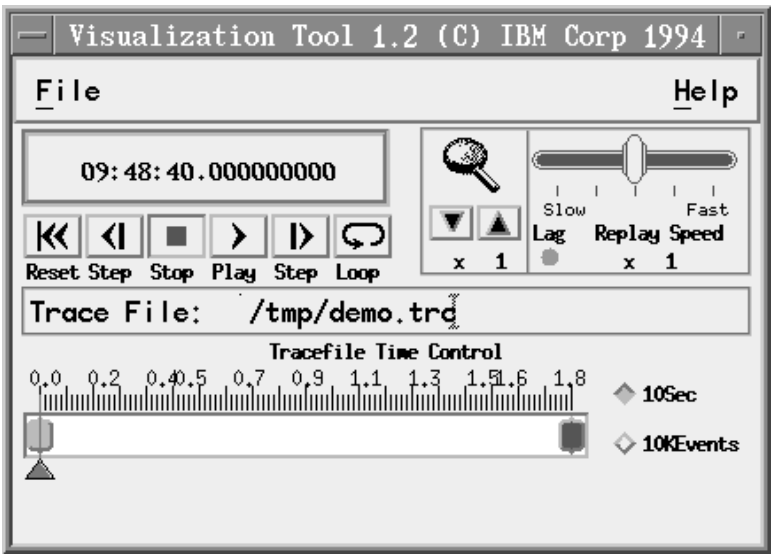


Figure 13. Trace Visualization Window

You use this window to monitor and control the playback of trace files. It consists of:

- A menu bar with the following options:
  - File
  - Help
- The *Application Time Display*. This display (just below the menu bar and on the left of the Trace Visualization window) shows the time recorded in the trace file during the application's execution. This time is displayed in hours, minutes, and seconds. Keep in mind that this display refers to the original processing time, and not to the trace file's playback time. If, for example, your program turns trace record generation off and on (as described in "Generating Trace Files" on page 106), the time shown in the Application Time Display will skip over the time during which trace record generation was set off.
- The *Basic Trace Visualization Controls*. These allow you to control the playback of your trace files and are similar to the controls you might find on a VCR or CD player. They are directly below the Application Time Display and, from left to right, are:
  - The *Reset Control Button*. This is used to return playback to the start (or an earlier point) in the trace file.
  - The *Step Back Control Button*. This allows you to step back a single trace record.
  - The *Stop Control Button*. This allows you to stop playback.
  - The *Play Control Button*. This allows you to start or resume playback.
  - The *Step Forward Control Button*. This allows you to step forward a single trace record.
  - The *Loop Control Button*. This allows you to continuously play a trace file or a portion of a trace file.

- The *Trace File Field*. This field is just below the Basic Trace Visualization Controls. It displays the full path name of the trace file you have selected for playback. You can also select a file for playback by entering it in this field.
- The *Trace File Time Control*. This control (at the bottom of the Trace Visualization window) contains a scale showing your current playback position either as elapsed time or as number of trace records. Two toggle buttons to the right of this control let you select whether elapsed time or number of trace records is presented. Labels on these toggle buttons indicate the value of units in the scale. As you play the trace file, a *Playback Position Line* moves through the scale to show your current playback position. You can also change the current playback position by dragging the small triangle at the base of the Playback Position Line to a new position within the Trace File Time Control.

  The Trace File Time Control also contains two range markers. The one to the left is the *start-of-range marker* and the one to the right is the *end-of-range marker*. You use these markers in conjunction with the Reset and Loop Control Buttons.
- The *Replay Speed Control*. This control is located towards the upper right of the Trace Visualization window, just below the menu bar. By adjusting the marker within it, you can change the speed of playback. By default, VT attempts to play back trace records at the same rate they were generated. If any of the open views cannot keep up with the speed at which VT is feeding them trace records, a *lag light* appears in the corner of the Replay Speed Control.
- The *Magnification Control*. This control (to the left of the Replay Speed Control) allows you to increase or decrease the amount of time represented in certain views. By varying the amount of time, you present more or less detail in these views. See “Step 8: Adjust View Magnification” on page 119 for more information on this control.

The following steps demonstrate how to play back trace files by leading you through all the controls on the Trace Visualization window. By following these steps, you can quickly master the simple controls and so be able to play and interpret your own trace files.

The intention of these steps is to touch on all the pertinent controls you can use when playing back a trace file. These steps do not indicate a specific order that must be followed when playing your own trace files. There are some obvious exceptions. For example, you must load a trace file before you can begin playback of it.

If you do not want to work through these steps because you are already familiar with VT, but do need to refresh your memory regarding some specific control, refer to the following table.

To:	Refer to:
load a trace file for playback	"Step 1: Load a Trace File for Playback" on page 114
start trace file playback	"Step 2: Start Playback of a Trace File" on page 115
return playback to the start (or an earlier point) in a trace file.	"Step 3: Return to an Earlier Point in the Trace File" on page 116
cycle playback through a trace file or a portion of a trace file.	"Step 4: Cycle Playback Through a Trace File" on page 117
reverse or advance stepping by a single trace record.	"Step 5: Stepping Playback" on page 117
adjust the speed of playback.	"Step 6: Adjust Playing Speed" on page 118
scroll back over view history buffers	"Step 7: Scrolling Over History Buffers" on page 118
adjust the level of detail shown in certain views.	"Step 8: Adjust View Magnification" on page 119

Notes:

1. In the steps that follow, use the left mouse button for all menu bar and control selections unless otherwise noted.
2. There is a sample trace file provided with VT in the directory */usr/lpp/vt/tracefiles*. It is called *vtsample.trc*. You can use it if you want to familiarize yourself with the controls described in the following steps, but have not yet generated any trace files of your own.
3. In the steps that follow, it is assumed that you have already opened the views appropriate to your needs as described in "Opening and Closing Views" on page 98. If you are new to VT and are following these steps to familiarize yourself with its controls, open the Interprocessor Communication, User Load Balance, and Source Code views for demonstration.

Step 1: Load a Trace File for Playback

Before you can play back a trace file, you need to load it. From the menu bar of the Trace Visualization window:

SELECT **File → Tracefile**

▲ The Trace File Selection Dialog window opens.

This window contains:

- A filter area showing the current search path
- A list of the directories in the current search path
- A list of the trace files in the current search path.

If the trace file you want to load is not in the current search path, you can specify a new search path. To do this:

FOCUS on the filter area.

TYPE IN the new search path.

PRESS **Filter**

▲ VT updates the list of directories and files accordingly.

To select a trace file for playback:

PLACE the cursor over its entry in the list of files.

PRESS the mouse button.

▲ The full path name of the trace file appears in the Trace File field.

PRESS **OK**

▲ VT loads the trace file and closes the Trace File Selection Dialog window.

Note: You can also load a trace file without using the Trace File Selection Dialog window. There are two ways to do this – from the Trace Visualization window, or when starting a VT session.

- To load a trace file from the Trace Visualization window:

  FOCUS on the Trace File field

  TYPE IN the full path name of the trace file.

  PRESS **Enter**

  ▲ VT loads the trace file for playback.

- To load a trace file when starting a VT session, use the **-tracefile** or **-tfile** flag on the **vt** command. For example, say you are starting a VT session and you want to load the sample trace file *vtsample.trc* from the directory */usr/lpp/vt/tracefiles*. You would:

  ENTER **vt -tracefile** */usr/lpp/vt/tracefiles/vtsample.trc*

  or

  vt -tfile */usr/lpp/vt/tracefiles/vtsample.trc*

  ▲ The Trace Visualization window and the View Selector window automatically open, marking the start of a VT session. In addition, VT loads the trace file *vtsample.trc* for playback.

Step 2: Start Playback of a Trace File

Once you have loaded a trace file you can start playing back its trace records. Before starting playback, you should open the views appropriate to your needs as described in “Opening and Closing Views” on page 98. To start playback:

PRESS the Play Control Button.

▲ Playback begins.

As a trace file plays back:

- All open views visualize the appropriate trace records generated when the parallel program was run.
- The Application Time Display shows the time recorded during the parallel program's execution.
- the Playback Position Line moves within the Trace File Time Control to show your current playback position. The Trace File Time Control shows your playback position in either elapsed time or number of trace records depending on which toggle button is pressed.

Playback continues until VT reaches the end-of-range marker or you press the Stop Control Button.

Note: On the **vt** command, you can use the **-go** flag to specify that playback should start immediately. You can only use the **-go** flag, however, if you also specify:

- the name of a trace file using the **-tracefile** or **-tfile** flag. See "Step 1: Load a Trace File for Playback" on page 114 for more information.
- the name of a configuration file using the **-configfile** or **-cfile** flags. See "Saving and Loading a VT Configuration File" on page 103 for more information.

For example, to start a VT session, load the sample trace file *mytracefile* and the configuration file *myconfigfile*, and start playback immediately:

ENTER **vt -tfile** *mytracefile* **-cfile** *myconfigfile* **-go**

Step 3: Return to an Earlier Point in the Trace File

The Reset control button returns playback to the start, or to an earlier spot in a trace file. Before you can return to an earlier position in a trace file, you must stop playback.

PRESS the Stop Control Button

▲ Playback stops at its current position in the trace file.

To return playback to the start of a trace file:

PRESS the Reset Control Button.

PRESS the Play Control Button.

▲ Playback returns to the start of the trace file. All views and the Application Time Display update accordingly.

By first adjusting the start-of-range marker in the Trace File time control, you can use the Reset control button to return to that marked spot rather than the start of the trace file. If you isolate an area of interest or concern within the trace file, you could do this to keep returning to the start of that area.

To return playback to an earlier spot in a trace file:

PRESS the Stop Control Button.

DRAG the start-of-range marker in the Trace File Time Control to the desired spot.

PRESS the Reset Control Button.

PRESS the Play Control Button.

▲ Playback returns to the spot indicated by the start-of-range marker. All Views and the Application Time Display update accordingly.

Step 4: Cycle Playback Through a Trace File

When VT reaches the end of a trace file, playback stops. If you would like playback to continuously cycle through a tracefile:

PRESS the Loop Control Button.

▲ When VT reaches the end of the tracefile, it returns to the start of the tracefile and resumes playback.

If you have isolated an area of interest or concern within a trace file, you might wish to cycle playback through just that area. You can do this by first adjusting the two range markers in the Trace File Time Control before pressing the Loop Control button.

If you wanted, for example, to cycle playback between the 40 and 60 percent marks on the Trace Visualization Time Control, you would:

PRESS the Stop Control Button.

DRAG the start-of-range marker to the 40 percent mark.

DRAG the end-of-range marker to the 60 percent mark.

DRAG the playback position line by its triangular base in between the two markers.

PRESS the Loop Control Button.

PRESS the Play Control Button.

▲ Playback cycles through just this portion of the trace file.

Step 5: Stepping Playback

The Step Back Control Button and the Step Forward Control Button let you reverse or advance playback over the previous or next trace record for which there is an open view. This is called *stepping*, and is useful when you have isolated an area of interest or concern within a trace file. During normal playback, VT attempts to simulate actual time by advancing the views after reading each second of the trace file. This can be a drawback when one second of a trace file contains many trace records and when there are large gaps between trace records. When one second of the trace file contains many trace records, stepping lets you visualize each trace record individually to provide you with more detailed information. Stepping is also useful when your trace file contains large gaps between trace records. For example, say trace record generation was turned off and on within your program as described in “Generating Trace Files” on page 106. If trace record generation was turned off for, say, five minutes of the program's run, VT does not automatically skip over that area of the trace file. During normal playback, you will have to wait five minutes before any of the views are updated. Stepping is then a convenient way of skipping to the next trace record. When you step, VT reads the previous or next trace record in the trace file. If none of your open views respond to that type of trace record, VT continues reading the trace file until it reaches a trace record for which there is an open view.

Before you can step through a trace file, you must stop playback.

PRESS the Stop Control Button

▲ Playback stops at its current position in the trace file.

To advance playback past the next trace record:

PRESS the Step Forward Control Button

▲ Playback advances past the next trace record for which there is an open view, and stops. All views, as well as the Application Time Display and the Trace File Time Control update accordingly.

To reverse playback over the previous trace record:

PRESS the Step Back Control Button

Step 6: Adjust Playing Speed

During normal playback, VT attempts to simulate actual time. By shortening the interval between view advances, you can make a trace file play faster. Similarly, by lengthening the interval between view advances, you can make a trace file play slower. VT's fastest speed is 100 times faster than real time. Its slowest speed is 100 times slower than real time.

To control the speed of playback by changing the interval between view advances:

DRAG the marker in the Replay Speed Control towards the desired speed.

PRESS the Play Control Button

▲ VT changes the interval between view advances as indicated by the number at the bottom of the Replay Speed Control.

Notes:

1. A dot may light in the corner of the Replay Speed Control during playback. This is the *lag light*, and means that the views cannot keep up with the number of trace records VT is feeding them.
2. You can also change the update rate, or time resolution, for individual views as described in “Adjusting a View's Display Rate and Colors” on page 101. Keep in mind, however, that a view cannot display information faster than VT advances it.

Step 7: Scrolling Over History Buffers

Views have *history buffers*. These are buffers that store trace events after they are updated off the view. During playback, you can scroll the view windows back over these history buffers to see the earlier information.

To scroll back over history buffers:

DRAG the playback position line by its triangular base to the left during playback.

▲ All the views scroll back accordingly. If you scroll past the start of a view's history buffer, it goes blank.

To resume playback:

PRESS the Play Control Button

▲ the views resume playing the current trace record information.

Step 8: Adjust View Magnification

The streaming views visualize information for a range of time, and can be manipulated using the magnification control to present more or less detailed information. For example, the Kernel Utilization (Graph) view (see page 137) is a streaming view. It shows kernel utilization for all processors over a range of time. By increasing or decreasing the range of time represented, you can present more or less detail in the view.

For example, say you are playing a trace file and the Kernel Utilization (Graph) view shows a large magnitude difference between two pixels. This represents a sudden increase in kernel utilization. Say each pixel represents one second of time. To understand the sudden increase in kernel utilization, you need a more detailed representation of those two seconds. By decreasing the range of time represented in the view, you can increase the level of detail for those two seconds. Increasing the magnification shows you more detail by decreasing the time represented. Decreasing the magnification, similarly, shows you less detail.

To adjust magnification:

PRESS the up or down arrow button in the Magnification Control as you desire.

▲ The number at the bottom of the Magnification Control shows the current magnification setting, and the views update accordingly.

Using VT for Performance Monitoring

Performance monitoring enables you to monitor the operational status and activity of any of the processor nodes of your POWERparallel system or RISC System/6000 network cluster. You do not need a trace file for this, because VT does the monitoring online. Each of the processor nodes has a Statistics Collector daemon that, upon request, supplies VT with samplings of AIX kernel statistics at a configurable interval. It is these statistics that you visualize during performance monitoring. This section describes how to:

- Select processor nodes for monitoring
- start monitoring the selected processor nodes
- change the interval at which AIX kernel statistics are sampled.

The Performance Monitor Window

When you select **File** → **Performance Monitor** from the Trace Visualization window, the Performance Monitor window and the PM View Selector window open.

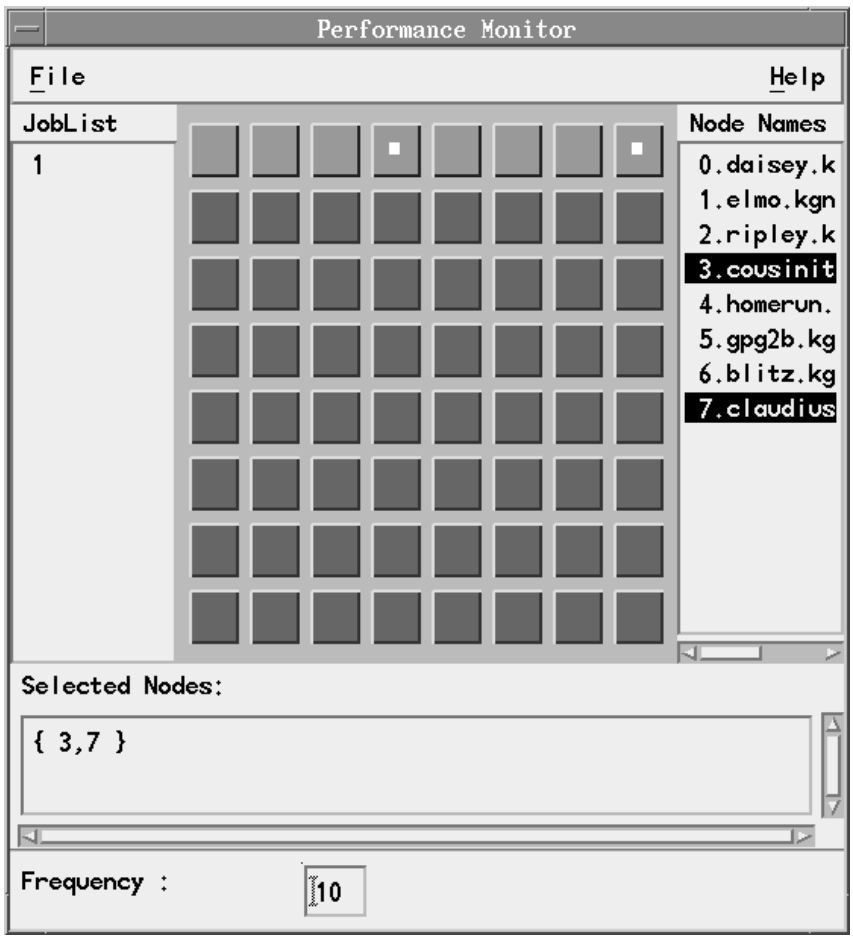


Figure 14. Performance Monitor Window

This window consists of:

- A *Job List*. This lists all the jobs currently running on the POWERparallel system by job number using data provided by the Resource Manager. Horizontal and vertical scroll bars let you view text outside the display area. If

you started VT with the **-norm** option (as you must for a RISC System/6000 network cluster), VT cannot track jobs and so cannot list them in this area.

- A *Node Matrix* This takes up the main part of the window. Each square on the matrix represents one of the processor nodes. The square's appearance indicates its status.

If the node appears in the matrix as a:	Its status is:
gray box	*unavailable*. You cannot select it for monitoring.
pink box	*available*. You can select it for monitoring.
green square within a pink box	*job running*. The processor node is currently running a job in the job list.
yellow box within a pink box	*selected for monitoring*. You have selected the processor node for monitoring.
yellow box within a green square within a pink box.	*active and selected* You have selected this active job for monitoring.

- A *Node List*. This lists all the processor nodes. Horizontal and vertical scroll bars let you view text outside the display area. VT gets this information from the POWERparallel system Resource Manager or, if you specified the **-norm** option when starting VT, from a host list file and from the LAN.
- A *Selected Nodes Area*. This area lists the processor nodes you select for performance monitoring.
- A *Frequency Control*. This control displays, and lets you reset the interval between which VT advances the views with new AIX kernel statistics from the Statistics Collector daemons.

The following steps demonstrate how to do performance monitoring on one or more processor nodes by leading you through the controls of the Performance Monitor window.

If you do not want to work through these steps because you are already familiar with VT, but do need to refresh your memory regarding some specific function of the Performance Monitor window, refer to the following table.

To:	Refer to:
Select individual processor nodes, or all the processor nodes associated with a particular job, for monitoring.	"Step 1: Select Processor Nodes for Monitoring" on page 122.
Start monitoring the selected processor nodes.	"Step 2: Start Monitoring" on page 122.
Change the interval between which VT updates the views with new AIX kernel statistics from the selected processor nodes.	"Step 3: Change Sampling Interval" on page 123.

Notes:

1. While the first two steps are required and must be performed in the order shown, the final step is optional.
2. In the steps that follow, it is assumed that you have already opened the views appropriate to your needs as described in "Opening and Closing Views" on page 98. If you are new to VT and are following these steps to familiarize yourself with its controls, open the Computation views for demonstration.

3. In the steps that follow, use the left mouse button for all menu bar and control selections.

Step 1: Select Processor Nodes for Monitoring

There are three ways to select processor nodes for monitoring. You can select them:

- From the Node Matrix
- from the Node List
- from the Job List.

To select a processor node from the Node Matrix.

PLACE the cursor on the square representing the processor node in the Node Matrix.

PRESS the mouse button.

▲ VT selects the processor node for monitoring. To show that it is selected, the processor node appears as a yellow box within a pink box. If the node is also active, it appears as a yellow box within a green square within a pink box. VT also adds the node's identifier to the Selected Nodes Area, and highlights the node in the node list.

To select processor nodes from the Node List.

PLACE the cursor on the processor node's entry in the Node List.

PRESS the mouse button.

▲ VT highlights the processor node's entry in the node list, and adds that node's identifier to the Selected Nodes Area. In the Node Matrix, the node appears as a yellow box within a pink box. If the node is also active, it appears as a yellow box within a green square within a pink box.

If you are using a POWERparallel system, you can also select all the processor nodes on which a particular job is running.

PLACE the cursor over that job's process identifier in the Job list.

PRESS the mouse button.

▲ VT highlights the job's process identifier, colors all the node squares associated with the job as a yellow box within a pink box, and lists the nodes in the Selected Nodes Area.

Note: If you are monitoring a RISC System/6000 network cluster, you cannot select processor nodes using the job list.

Step 2: Start Monitoring

Before you can start monitoring the selected processor nodes, you must open views as described in “Opening and Closing Views” on page 98. To start monitoring the performance of selected nodes:

SELECT **File → Monitor**

▲ The Statistics Collector daemon(s) on the selected processor node(s) begin sending samplings of AIX kernel statistics to VT. All open views begin visualizing these statistics.

Notes:

1. If communication between VT and a selected node cannot be established by the network, it may take a few minutes for the network to respond with an error.
2. If you select additional processor nodes after you have already started monitoring, you must stop and restart monitoring. **File → Monitor** is a toggle switch, so you stop monitoring the same way you started it:

   SELECT File → Monitor

Step 3: Change Sampling Interval

As you monitor the selected processor nodes, VT advances the views every second with a new sampling of AIX kernel statistics from the Statistics Collector daemons. You can change the frequency at which VT advances the views by resetting the sampling interval. You should increase the sampling interval if you have many performance monitor views open, or if there is a significant network delay in communicating with one or more of the nodes. If you have already started monitoring the selected processor nodes, you must stop monitoring before you reset the sampling interval.

SELECT File → Monitor

To reset the sampling interval:

FOCUS on the Frequency Control. The area shows the default as 10 seconds.

TYPE IN the new sampling interval in seconds.

PRESS Enter

▲ VT advances the views according to the new interval setting.

View Descriptions

This reference section describes each of the views you can open for trace visualization or performance monitoring. It is divided in the five view categories.

- Communication/Program (for trace visualization only)
- Computation
- Disk
- Network
- System

For those new to VT, a recommended initial selection of views is:

- Message Status Matrix (a Communication/Program view)
- User Load Balance (a Computation view)
- Source Code (a Communication/Program view)

You can then select additional views as the need arises. Using the left mouse button on a view displays additional information regarding events where the cursor is positioned. Using the right mouse button allows you to customize individual views.

View Types: There are two types of views – instantaneous and streaming. Instantaneous views present information for a specific point in time, while streaming views represent a range of time. On the streaming views, a vertical line drawn towards the right of the view represents the current point of trace playback or, if you are doing online monitoring, the current point in time.

Because the duration of communication events can vary greatly, VT does the following to ensure that all communication events are reasonably displayed in streaming views regardless of their duration or the current level of magnification:

- The duration of some events may be so short that, given the current level of magnification, the event would represent less than one column of pixels in a streaming view. In these situations, VT represents the event with one column of pixels. This ensures that VT shows all communication events. To get a more accurate representation of the event's duration, you could increase the magnification.
- The duration of some events may be so long that, given the current level of magnification, the block formed to represent it would take up too much room on the streaming view and render it less understandable. In these situations, VT represents the time period with the maximum-sized block it allows. VT fills the events that span this time period with a hashed pattern to show that it is not in proportion with the rest of the view. This is called *time compression*. To get a more accurate representation of the event's duration, you could decrease the magnification.

Communication/Program Views

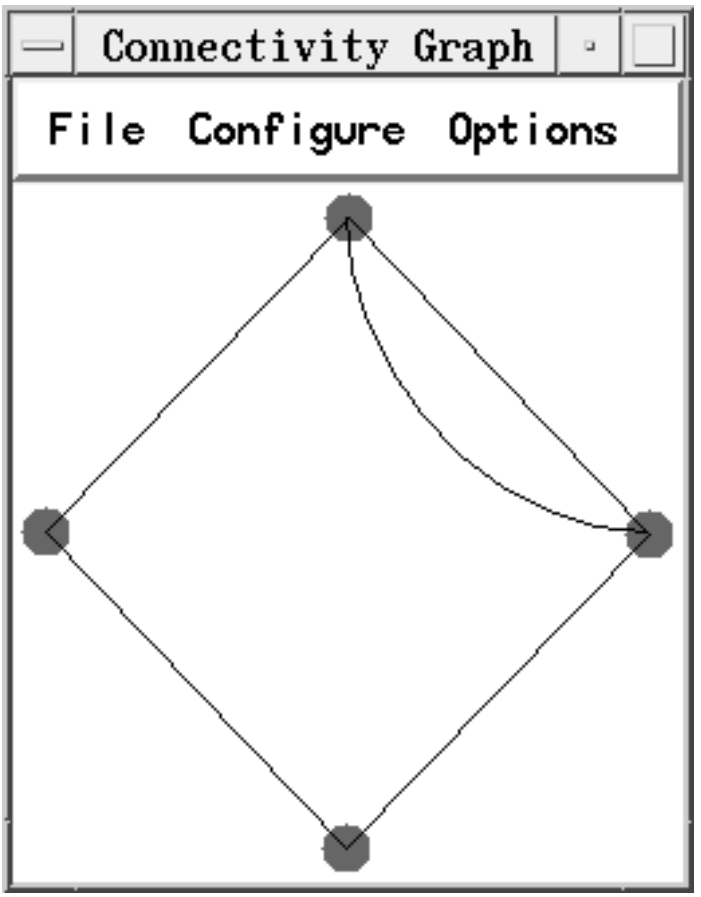
The four views in this category visualize communication events between processor nodes of your system, and information about your parallel program. The views in this category are for trace visualization only, and should not be opened once playback of the trace file has begun. They respond to the following types of trace records:

- Message Passing
- Application Markers

For more information on the trace records included in each type, refer to “Types of Trace Records” on page 105.

Connectivity Graph

This view visualizes message passing and collective communication events. The view uses a small circle to represent each processor node on which your program was run. Message passing events appear as an arc joining two of the circles. Nodes that are participating in collective communication are connected by straight lines. This is an instantaneous view.



Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any one of the small circles in the view.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- number of messages sent from that node
- number of messages received by that node.

The window remains open only while you hold the mouse button down.

Toggling Between Instantaneous and Cumulative Presentation: This view lets you toggle between an instantaneous and a cumulative presentation of the message passing information. An instantaneous presentation is the default – the arc representing the message passing event is visible only for the duration of the communication. With a cumulative presentation, the arcs remain visible past the completion of the communication. A cumulative presentation lets you see the overall communication pattern among the processor nodes.

If you want a cumulative presentation of the message passing information:

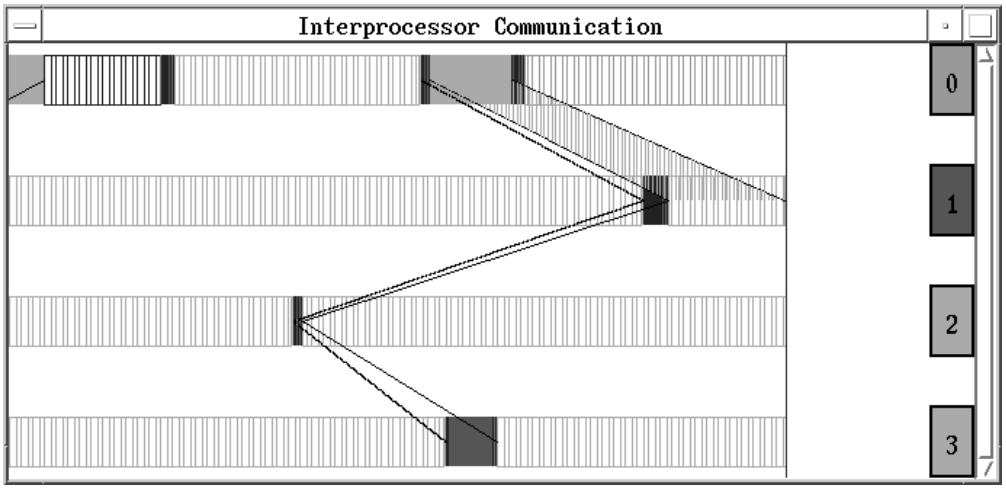
SELECT Options → Cumulative

Interprocessor Communication

This view uses a bar chart to visualize the type and duration of communication events. Each bar represents a processor node on which your program was run, and the chart's horizontal axis represents a range of time. A label to the right of each bar shows the node number, and is colored based on the current state of that node. A current time line is drawn down the view's 90 percent mark. Each bar in the chart will be made up of a number of colored blocks. Each block represents a communication event involving the processor. The size of the block represents the event's duration, and its color indicates the type of event. For example, blocking sends are shown in one color, non-blocking sends in another, broadcasts in another, and so on. When messages are sent between processor nodes, a message line is drawn between the appropriate bars in the chart and the node labels light up. When nodes are involved in collective communication, a polygon is drawn covering the collective communication events.

Lines are drawn when the communication is known to have completed. They are drawn from the start of the event which initiated the communication to the end of the event which completed it. Thus, lines are drawn from the start of blocking or non-blocking sends to the ends of blocking receives, waits and status events. Lines are only drawn when the wait or status for any non-blocking calls involved in the communication have completed. This is a streaming view.

Note: If you turned tracing off and on from your application program (see “Step 1: Control Trace Record Generation Within the Program” on page 106), and the resulting trace file does not contain both the send and receive for each communication event, the message lines drawn on the view may not be valid. Unless there is a corresponding receive trace record for each send, there is a danger that VT could match the wrong receive to a send – drawing a misleading message line.



Using the Search Feature: This view has a search feature that helps you to easily examine the communication events and processor nodes you are interested in. For example, say you are interested in any blocking sends involving node 2 or 6. You would:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Search**

The Search window opens, and VT draws a vertical search line down the view's 50 percent mark. You are now in search mode.

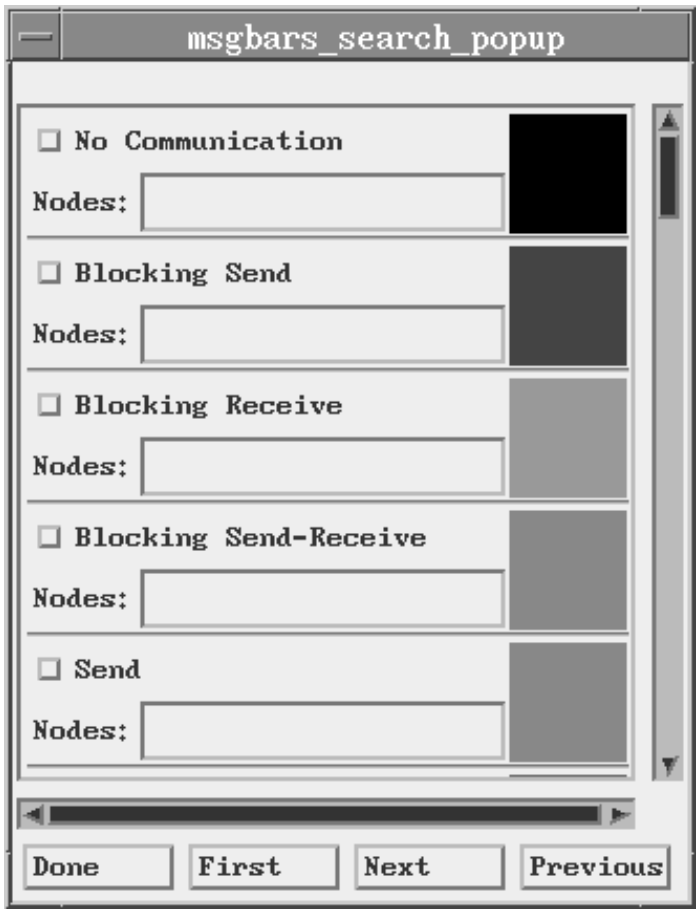


Figure 15. The Search window

The Search window contains a separate area for each type of communication event. Each area contains a text entry field and a selection button. The color shown in each area corresponds to the color used for that type of communication event in the view.

PRESS the blocking send button.

FOCUS on the text entry field for blocking sends.

TYPE IN 2,6

To see the first blocking send involving the two nodes:

PRESS **First**

▲ The first blocking send involving the two nodes is lined up under the Search line. In search mode, the processor labels are colored to represent what's under the search line.

To see the next blocking send involving the two nodes:

PRESS **Next**

To see the previous blocking send involving the two nodes:

PRESS **Previous**

To close the Search window and return to play mode.

PRESS **Done**

The search feature can perform a number of searches simultaneously. For example, you could set up a search for blocking sends involving node 2 or 6 *and* a search for receives involving node 9, 10, or 11. Pressing **First**, **Next**, or **Previous** shows you the first, next, or previous event meeting any of the specified search requirements.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any one of the processor labels.

PRESS the left mouse button.

▲ A window opens displaying the:

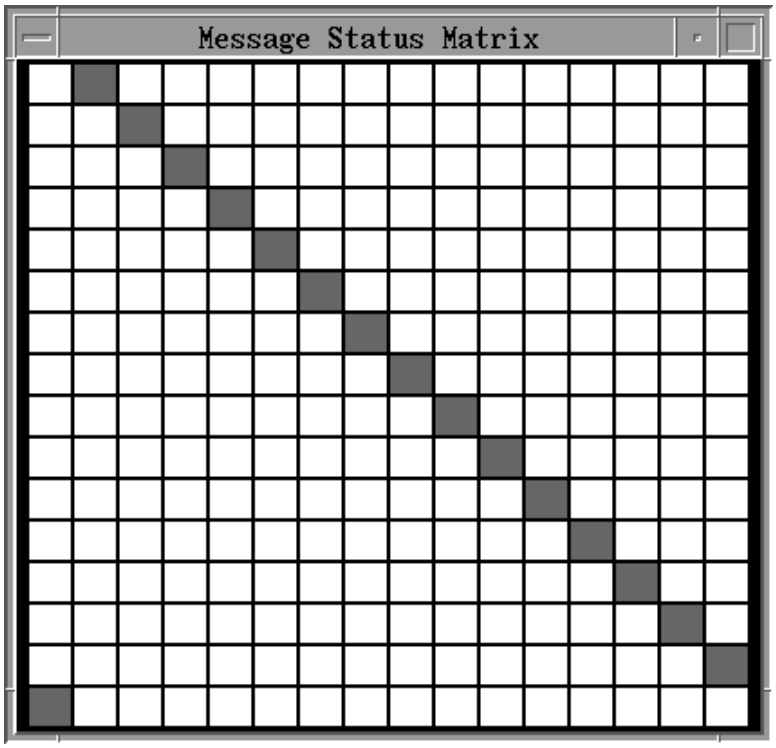
- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- the type of communication event
- the time the event occurred.

During regular play mode, the information displayed is for the communication event under the current time line. During search mode, the information displayed is for the communication event under the search line. The window remains open only while you hold the mouse button down.

Customizing the View's Colors: As described in “Adjusting a View's Display Rate and Colors” on page 101, the appearance of a view – the colors it uses – is determined by its display spectrum. The Communication display spectrum is designed especially for use with this view. Each color in the spectrum is a resource you can customize using the *.Xdefaults* file. This enables you to specify the exact color used to represent each type of event in the view. For more information, see “VT Resource Variables” on page 220.

Message Status Matrix

This view uses a grid to visualize messages sent between processor nodes. Each processor node has both a row and a column on the table. The rows represent when processor nodes send a message, and the columns represent when processor nodes receive a message. The rectangle intersections on the grid represent the message path between the sending node (the row) and the receiving node (the column). As you play back your trace file, the message path rectangles light up from the time the message begins to be sent to the time the receive completes. This is an instantaneous view.



Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any of the rectangles representing message paths between processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number of the sending node.
- node number of the receiving node.
- number of messages sent.

The window remains open only while you hold the mouse button down.

Toggling Between Instantaneous and Cumulative Presentation: This view lets you toggle between an instantaneous and a cumulative presentation of the message passing information. An instantaneous presentation is the default – the message path rectangles stay lit only for the duration of the message passing event. With a cumulative presentation, the message path rectangles remain lit past the completion of the event. A cumulative presentation lets you see the overall communication pattern among the processor nodes.

If you want a cumulative presentation of the message passing information:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

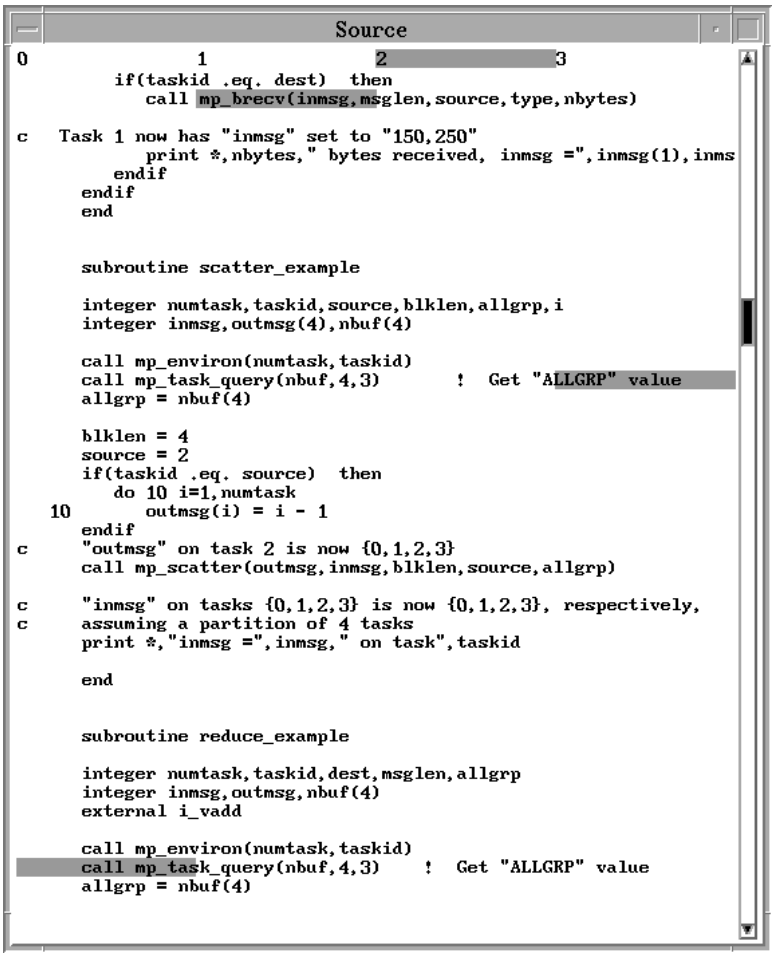
▲ A selection menu appears.

SELECT **Cumulative**

Source Code

This view shows you the C, C++ or FORTRAN source code of the program associated with the most recent trace event. A series of colored bars across the top of the display represent the different program tasks. As you play back the trace file, the bars move through the code to show you each task's position in relation to the source. To be more specific, for each time a task entered into a communication function such as a blocking send, an environment initialization, or an application marker call, its bar moves to that line in the source code. When there are several source files for the executable, the view will switch to show the new source as soon as one program task goes into it.

You can only use the Source Code view when playing back a tracefile generated from a SPMD application. This view is not valid when playing back a tracefile generated from a MPMD application. In order to use this view, you must also have compiled your program with the **-g** flag, and the Source Code files must be in your current directory or in a special search path used by the view. "Adding and Deleting Paths to the Source" on page 132 describes how to add directories to this source code search path. This is an instantaneous view.



Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any one of the bars representing a program task.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- task identifier
- list of tasks at that line in the source code file
- name of the source code file
- line number you are at in the source code file.

Using the Automatic Scrolling Capability: This view has an automatic scrolling capability that automatically scrolls the view to include the most recent trace event. If this capability is off, you must manually scroll the view. To toggle the automatic scrolling capability on and off:

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Parameters**

▲ The view's Parameters Window opens.

PRESS the autoscroll toggle button on the Parameter Window.

PRESS **Apply**

PRESS **OK**

Toggling Between An Instantaneous and Cumulative Presentation: The view lets you toggle between instantaneous and cumulative presentation of the source code's relation to trace events. By default, the view shows an instantaneous presentation – the colored bars representing program tasks move through the source to show each task's position. Each time the task enters into a communication function, its bar moves forward to the associated line in the source. Its bar colors that line only until the task enters into the next communication function. In a cumulative presentation, once a line of the source is colored by a task bar, it remains colored. In this way, each task leaves a trail through the source.

If you want a cumulative presentation of the Source Code view:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Cumulative**

Adding and Deleting Paths to the Source: In order to display a source code file in this view, it must be in your current directory or in a special source code search path. When you start VT, you can indicate a search path to a program's source code using the **-spath** flag. See Table 9 on page 98.

You can also modify your source code search path from the Source Code view window. To do this:

PRESS the right mouse button

SELECT Source

▲ The Select Source Files window opens.

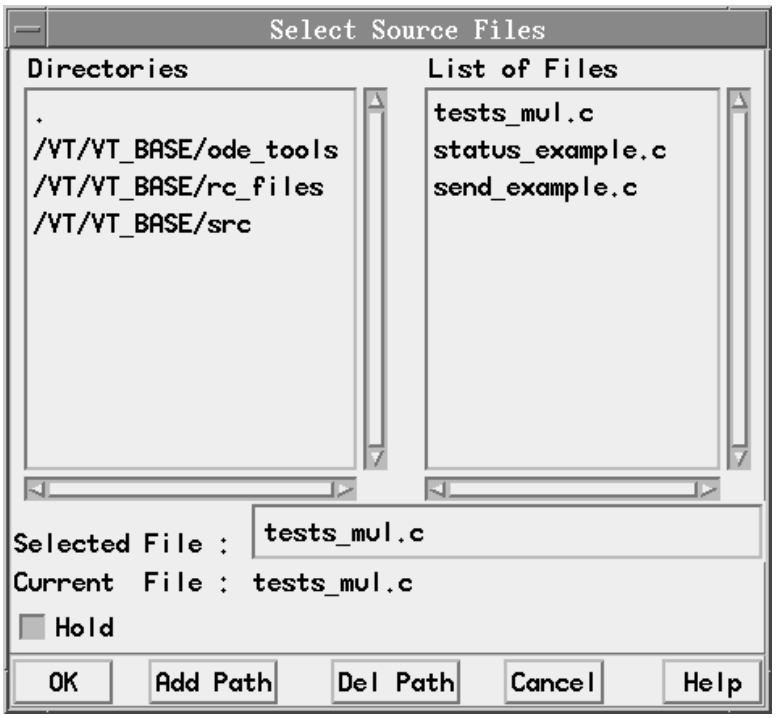


Figure 16. Select Source Files Window

This window contains an area listing all the directories in your source code search path.

To add a new directory to the source code search path:

PLACE the mouse cursor over the name of one of the directories listed.

PRESS the left mouse button.

▲ VT highlights the directory's entry in the list to show that it is selected.

PRESS Add Path

▲ The Add Path window opens.

Figure 17. Add Path Window

FOCUS on this window's text entry field.

TYPE IN the directory path you want to add.

PRESS either the **Add Before** or **Add After** toggle button, depending on whether you want the new directory path to be added to your source code search path before or after the selected directory.

PRESS **OK**

▲ The Add Path window closes. The directory search path you specified is added to the list of directories in your source code search path.

To delete a directory's entry from the source code search path:

PLACE the mouse cursor over the name of one of the directories listed.

PRESS the left mouse button.

▲ VT highlights the directory's entry in the list to show that it is selected.

PRESS **Del Path**

▲ A pop-up window opens, asking if you are sure.

PRESS **OK** on the pop-up.

▲ The pop-up window closes, and VT deletes the selected directory from your source code search path.

To close the Select Source Files window:

PRESS **OK**

Selecting a Source to Display in the View: You can select any source in your source code search path and have it displayed in the Source Code view. To do this:

PRESS the right mouse button

SELECT **Source**

▲ The Select Source Files window (see Figure 16 on page 133) opens. The names of all the source code files in your source code search path are displayed in the List of Files area.

PLACE the mouse cursor over the name of the file you want displayed.

PRESS the right mouse button.

▲ The source code file name appears in the Selected File field.

PRESS **OK**

▲ The Select Source Files window closes and the view displays the selected source file.

Holding the Current Source File: When there are several source files for the executable, the view will switch to show the next source as soon as one program task goes into it. If you want to stay with the current source file, and not have the view switch to others:

PRESS the right mouse button

SELECT **Source**

▲ The Select Source Files window opens.

PRESS the **Hold** toggle button in the lower left corner of this window.

PRESS **OK**

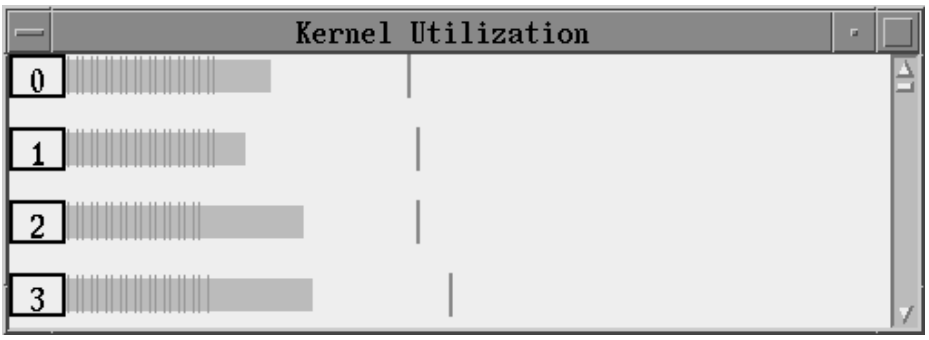
▲ The Select Source Files window closes and the view now stays with the current source file.

Computation Views

The views in this category visualize information regarding the utilization of the processor nodes running a particular program. The views in this category are for trace visualization and performance monitoring, and they respond to AIX kernel statistics trace records. For more information on, and a listing of, the AIX kernel statistic trace records, see “Types of Trace Records” on page 105.

Kernel Utilization (Bar Chart)

This view uses a bar chart to visualize kernel utilization for each of the processor nodes. Each node is represented as a bar on the chart. As you play back the trace file or monitor nodes online, the length of the bars change to show their relative value of kernel utilization. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous kernel utilization. The hatched fill pattern is the average kernel utilization. In addition, VT draws a vertical line to the right of each bar to show the highest level of kernel utilization so far reached by each node. This is an instantaneous view.



Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any one of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous value of kernel utilization
- average value of kernel utilization
- maximum value of kernel utilization

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents kernel utilization as a percentage of 100. You can adjust this so that the view represents kernel utilization as a percentage of some other value. For example, say you ran your program on four processor nodes – none of which ever exceeded 10 percent kernel utilization. You might want to adjust the view so that it represents kernel utilization as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

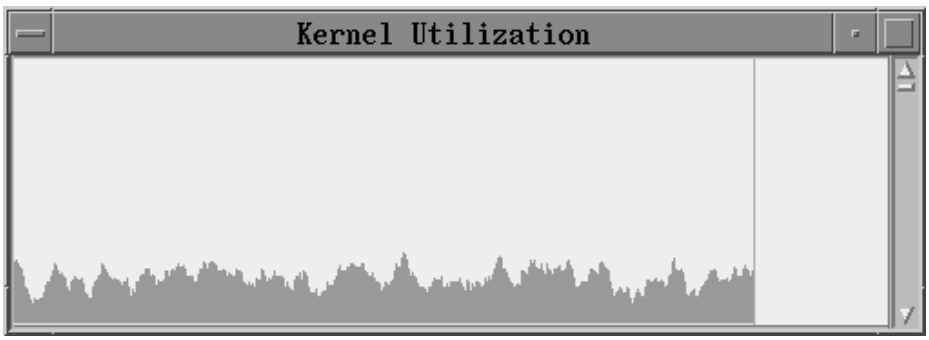
PRESS **Apply**

▲ The view calibrates to the new maximum value.

Kernel Utilization (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize kernel utilization for the processor nodes. This view has two display modes – *aggregate* and *individual*. By default the view is in aggregate display mode – it uses a single strip graph. A line on this graph shows the average value of kernel utilization across all processor nodes. Individual display mode shows a separate strip graph for each processor node's kernel utilization. When in this mode, each graph

contains two lines – one showing the individual processor node's kernel utilization and, for comparison, one showing the aggregate kernel utilization. To distinguish between the two, the area between the individual kernel utilization line and the graph's horizontal axis has a solid fill. This is a streaming view.



Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on a line representing either aggregate or individual kernel utilization.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate kernel utilization:	If the cursor is on a line representing an individual node's kernel utilization:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate kernel utilization at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate kernel utilization at that point in time. • the individual processor node's kernel utilization at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

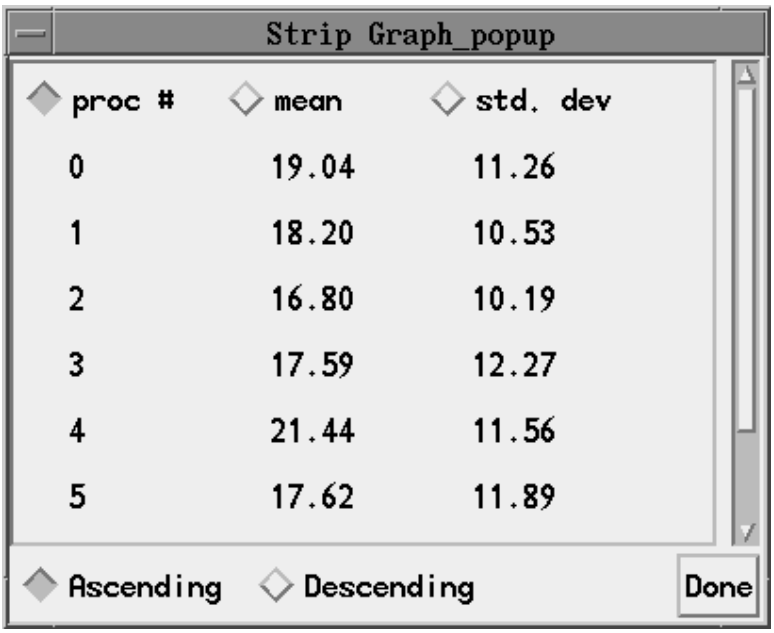
PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT Show Statistics

▲ The Statistics Window opens.



The Statistics Window lists the mean and standard deviation for each processor node. The toggle buttons enable you to sort the individual views in ascending or descending order according to processor node id, mean, and standard deviation. To close the Statistics Window:

PRESS Done

Processor Wait (Bar Chart)

This view uses a bar chart to visualize the percentage of time the processor node spent waiting for some resource to become available. Each node is represented as a bar on the chart. As you play back the trace file or monitor nodes online, the length of the bars change to show the relative percentage of processor wait time. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous wait time. The hatched fill pattern is the average wait time. In addition, VT draws a vertical line to the right of each bar to show the highest percentage of processor wait time so far reached by each node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any one of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous value of processor wait time

- average value of processor wait time
- maximum value of processor wait time

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents processor wait time as a percentage of 100. You can adjust this so that the view represents processor wait time as a percentage of some other value. Say you ran your program on four processor nodes, and the processor wait time did not exceed 10 percent on any of them. You might want to adjust the view so that it represents processor wait time as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value.

Processor Wait (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize the percentage of time the processor nodes spent waiting for some resource to become available. This view has two display modes – *aggregate* and *individual*. By default the view is in aggregate display mode – it uses a single strip graph. A line on this graph shows the average percent of wait time across all processor nodes. Individual display mode shows a separate strip graph for each processor node's wait time. When in this mode, each graph contains two lines – one showing the individual processor node's wait time and, for comparison, one showing the aggregate processor wait time. To distinguish between the two, the area between the individual kernel utilization line and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on a line representing either aggregate or individual processor wait time.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate processor wait time:	If the cursor is on a line representing an individual node's wait time:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate processor wait time at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate processor wait time at that point in time. • the individual processor node's wait time at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Show Statistics**

▲ The Statistics Window opens. This window is shown on page 138.

Processor Idle (Bar Chart)

This view uses a bar chart to visualize the percentage of time the processor nodes spent idle. Each node is represented as a bar on the chart. As you play back the trace file or monitor nodes online, the length of the bars change to show the relative percentage of processor idle time. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous idle time. The hatched fill pattern is the average idle time. In addition, VT draws a vertical line to the right of each bar to show the highest percentage of processor idle time so far reached by each node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any one of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous value of processor idle time
- average value of processor idle time
- maximum value of processor idle time

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents processor idle time as a percentage of 100. You can adjust this so that the view represents processor idle time as a percentage of some other value. Say you ran your program on four processor nodes, and the processor idle time did not exceed 10 percent on any of them. You might want to adjust the view so that it represents processor idle time as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value.

Processor Idle (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize the percentage of time processor nodes spent idle. This view has two display modes – *aggregate* and *individual*. By default the view is in aggregate display mode – it uses a single strip graph. A line on this graph shows the average percent of idle time across all processor nodes. Individual display mode shows a separate strip graph for each processor node's idle time. When in this mode, each graph contains two lines – one showing the individual processor node's idle time and, for comparison, one showing the aggregate processor idle time. To distinguish between the two, the area between the individual kernel utilization line and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on a line representing either aggregate or individual processor idle time.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate processor idle time:	If the cursor is on a line representing an individual node's idle time:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate processor idle time at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate processor idle time at that point in time. • the individual processor node's idle time at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Show Statistics**

▲ The Statistics Window opens. This window is shown on page 138.

User Utilization (Bar Chart)

This view uses a bar chart to visualize CPU utilization for each of the processor nodes. Each node is represented as a bar on the chart. As you play back the trace file or monitor nodes online, the length of the bars change to show their relative value of CPU utilization. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous CPU utilization. The hatched fill pattern is the average CPU utilization. In addition, VT draws a vertical line to the right of each bar to show the highest level of CPU utilization so far reached by each node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any one of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous value of CPU utilization
- average value of CPU utilization
- maximum value of CPU utilization

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents processor utilization as a percentage of 100. You can adjust this so that the view represents processor utilization as a percentage of some other value. For example, say you ran your program on four processor nodes – none of which ever exceeded 10 percent utilization. You might want to adjust the view so that it represents processor utilization as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value

User Utilization (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize CPU utilization for the processor nodes. This view has two display modes – *aggregate* and *individual*. By default the view is in aggregate display mode – it uses a single strip graph. A line on this graph shows the average value of CPU utilization across all processor nodes. Individual display mode shows a separate strip graph for each processor node's CPU utilization. When in this mode, each graph contains two lines – one showing the individual processor node's CPU utilization and, for comparison, one showing the aggregate CPU utilization. To distinguish between the two, the area between the individual CPU utilization line and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on a line representing either aggregate or individual CPU utilization.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate CPU Utilization:	If the cursor is on a line representing an individual node's CPU utilization:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate CPU utilization at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate CPU utilization at that point in time. • the individual processor node's CPU utilization at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

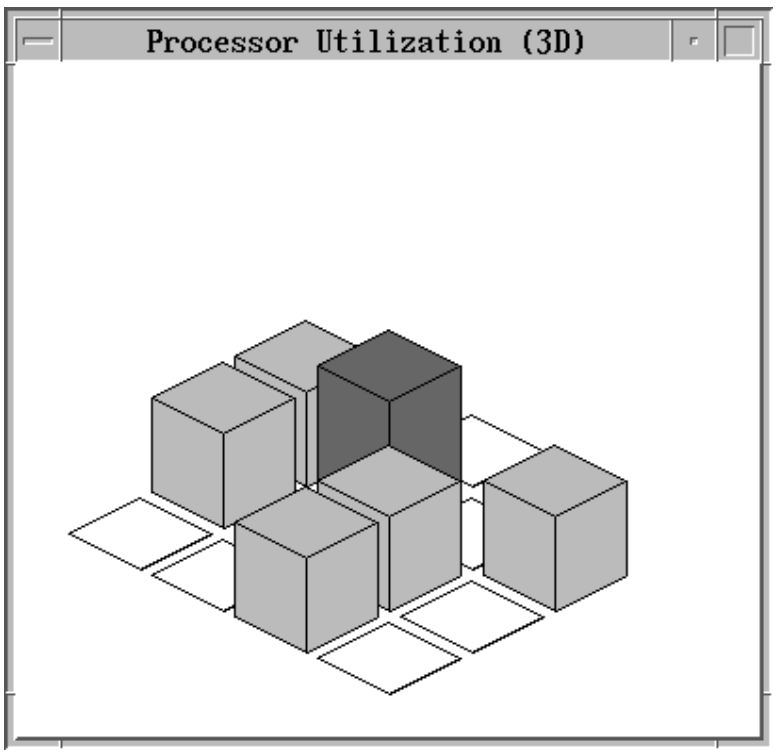
▲ A selection menu appears.

SELECT Show Statistics

▲ The Statistics Window opens. This window is shown on page 138.

Processor Utilization (3D Bar Chart)

This view is similar to the user utilization bar chart. It too visualizes CPU utilization for each of the processor nodes during the program's run. In this view, however, the processor nodes are laid out as bars in a two-dimensional grid and their CPU utilization values raise the bars up along the z-axis. As the bars rise up along the z-axis, they appear in different colors according to your display spectrum. This is an instantaneous view, and is for trace visualization only.



Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any one of the bars representing a processor node.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous CPU utilization

Changing the Display Angle: You can change the angle at which the view displays the 3-D Bar Chart. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Parameters**

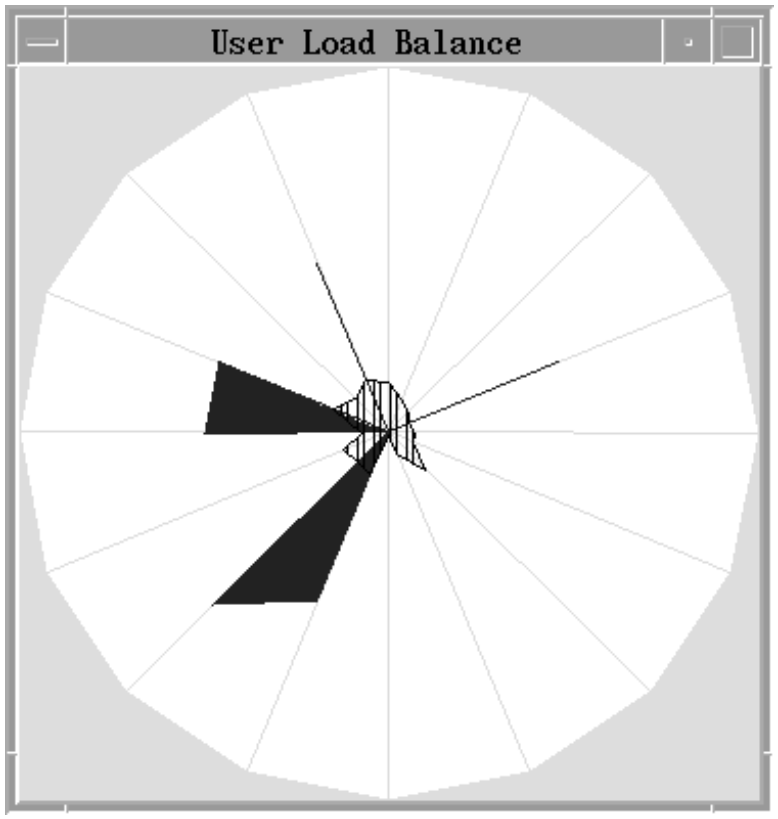
▲ The view's Parameters window opens.

FOCUS on the angle field

TYPE IN 0, 1, or 2. Specifying 0 gives you a side view of the chart, while specifying 2 angles the chart as if you were looking down on it. 1 is in between these two extremes.

User Load Balance

This view uses three overlapping polygons to show CPU utilization for each of the processor nodes, and the overall processor load balance. This is an instantaneous view.



The largest of the polygons represents 100 percent utilization for all of the processor nodes. Within this polygon, VT draws each processor node as a spoke starting at the polygon's center and extending out to the rim.

VT draws the second polygon inside the first. This polygon represents the instantaneous CPU utilization for each of the processor nodes. On each node's spoke, VT draws a point which represents the current CPU utilization for that node. VT then connects the points to form a polygon with a solid fill pattern. The more regular the polygon, the better your processor load balance.

The third polygon is similar to the second. It shows the average CPU utilization, however, and has a hatched fill pattern.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any one of the spokes representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- the instantaneous value of CPU utilization
- the average value of CPU utilization.

The window remains open only while you hold the mouse button down.

Disk Views

The six views in this category visualize the number of times processes acquire information from, or send information to, the systems' hard disks. These views are for trace visualization and performance monitoring, and respond to AIX kernel statistic trace records. For more information on AIX kernel statistic trace records, refer to “Types of Trace Records” on page 105.

Disk Reads (Bar Chart)

This view uses a bar chart to visualize disk reads – the number of times processes acquire information from the systems' hard disks. This refers to reads of cached information rather than access to the actual physical device. Each node is represented by a bar on the chart. If you are using this view for trace visualization, these are the processor nodes that ran your program. If you are using this view for performance monitoring, these are the selected nodes. The length of the bars change to show the number of disk reads. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous number of disk reads. The hatched fill pattern is the average number of disk reads. In addition, VT draws a vertical line to the right of each bar to show the highest instantaneous number of disk reads so far reached for each processor node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous number of disk reads
- average number of disk reads
- maximum number of disk reads so far recorded

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents disk reads as a percentage of 100. You can adjust this so that the view represents disk reads as a percentage of some other value. For example, say the amount of disk reads never exceeds 10 percent. You might want to adjust the view so that it represents disk reads as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value.

Disk Reads (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize disk reads – the number of times processes acquire information from the systems' hard disks. This refers to reads of cached information rather than the actual physical device. If you are using this view for trace visualization, the processor nodes depicted are the ones that ran your program. If you are using this view for performance monitoring, these are the selected nodes.

This view has two display modes – aggregate and individual. By default, the view is in aggregate display mode – it uses a single strip graph to represent the average number of disk reads across all processor nodes. Individual display mode shows a separate graph for each processor node's disk reads. When in this mode, each graph contains two lines – one showing the individual processor node's disk reads, and, for comparison, one showing the aggregate number of disk reads. To distinguish between the two, the area between the individual disk read line and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on the line representing either aggregate or individual disk reads.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate disk reads:	If the cursor is on a line representing an individual node's disk reads:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of disk reads at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of disk reads at that point in time. • the individual processor node's number of disk reads at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Show Statistics**

▲ The Statistics Window opens. This window is shown on page 138.

Disk Transfers (Bar Chart)

This view uses a bar chart to visualize disk transfers – the number of times the system transfers blocks of read/write data to and from the hard disks. Each node is represented by a bar on the chart. If you are using this view for trace visualization, these are the processor nodes that ran your program. If you are using this view for performance monitoring, these are the selected nodes. The length of the bars change to show the number of disk transfers. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous number of disk transfers. The hatched fill pattern is the average number of disk transfers. In addition, VT draws a vertical line to the right of each bar to show the highest instantaneous number of disk transfers so far reached for each processor node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous number of disk transfers
- average number of disk transfers
- maximum number of disk transfers so far recorded

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents disk transfers as a percentage of 100. You can adjust this so that the view represents disk transfers as a percentage of some other value. For example say the amount

of disk transfers never exceeds 10 percent. You might want to adjust the view so that it represents disk transfers as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value.

Disk Transfers (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize disk transfers – the number of times the system transfers blocks of read/write data to and from the hard disks. If you are using this view for trace visualization, the processor nodes depicted are the ones that ran your program. If you are using this view for performance monitoring, they are the selected nodes.

This view has two display modes – aggregate and individual. By default, the view is in aggregate display mode – it uses a single strip graph to represent the average number of disk transfers across all processor nodes. Individual display mode shows a separate graph for each processor node's disk transfers. When in this mode, each graph contains two lines – one showing the individual processor node's disk transfers, and, for comparison, one showing the aggregate number of disk transfers. To distinguish between the two, the area between the individual disk transfers line and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on the line representing either aggregate or individual disk transfers.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate disk transfers:	If the cursor is on a line representing an individual node's disk transfers:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of disk transfers at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of disk transfers at that point in time. • the individual processor node's number of disk transfers at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Show Statistics**

▲ The Statistics Window opens. This window is shown on page 138.

Disk Writes (Bar Chart)

This view uses a bar chart to visualize disk writes – the number of times processes write information to disk. Each node is represented by a bar on the chart. If you are using this view for trace visualization, these are the processor nodes that ran your program. If you are using this view for performance monitoring, these are the selected nodes. The length of the bars change to show the number of disk writes. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous number of disk writes. The hatched fill pattern is the average number of disk writes. In addition, VT draws a vertical line to the right of each bar to show the highest instantaneous number of disk writes so far reached for each processor node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous number of disk writes
- average number of disk writes
- maximum number of disk writes so far recorded

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents the number of disk writes as a percentage of 100. You can adjust this so that the view represents disk writes as a percentage of some other value. For example, say the number of disk writes never exceeds 10 percent. You might want to adjust the view so that it represents disk writes as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value.

Disk Writes (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize disk writes – the number of times processes write information to disk. If you are using this view for trace visualization, the processor nodes it depicts are the ones that ran your program. If you are using this view for performance monitoring, they are the selected nodes.

This view has two display modes – aggregate and individual. By default, the view is in aggregate display mode – it uses a single strip graph to represent the average number of disk writes across all processor nodes. Individual display mode shows a separate graph for each processor node's disk writes. When in this mode, each graph contains two lines – one showing the individual processor node's disk writes, and, for comparison, one showing the aggregate number of disk writes. To distinguish between the two, the area between the individual disk writes line and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on the line representing either aggregate or individual disk writes.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate disk writes:	If the cursor is on a line representing an individual node's disk writes:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of disk writes at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of disk writes at that point in time. • the individual processor node's number of disk writes at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT Individual

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT Show Statistics

▲ The Statistics Window opens. This window is shown on page 138.

Network Views

The four views in this category visualize the number of TCP/IP packets sent or received. These views are for trace visualization and performance monitoring, and respond to AIX kernel statistic trace records. For more information on AIX kernel statistic trace records, refer to “Types of Trace Records” on page 105.

Packets Received (Bar Chart)

This view uses a bar chart to visualize the number of TCP/IP packets received by processor nodes. Each node is represented by a bar on the chart. If you are using this view for trace visualization, these are the processor nodes that ran your program. If you are using this view for performance monitoring, these are the selected nodes. The length of the bars change to show the number of packets received. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous number of packets received. The hatched fill pattern is the average number of packets received. In addition, VT draws a vertical line to the right of each bar to show the highest instantaneous number of packets received so far reached by each processor node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous number of packets received
- average number of packets received
- maximum instantaneous number of packets so far received

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents packets received as a percentage of 100. You can adjust this so that the view represents packets received as a percentage of some other value. For example, say the number of packets received never exceeds 10 percent. You might want to adjust

the view so that it represents packets received as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value.

Packets Received (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize the number of TCP/IP packets received by processor nodes. If you are using this view for trace visualization, the processor nodes it depicts are the ones that ran your program. If you are using this view for performance monitoring, these are the selected nodes.

This view has two display modes – aggregate and individual. By default, the view is in aggregate display mode – it uses a single strip graph to represent the average number of packets received across all processor nodes. Individual display mode shows a separate graph for each processor node's packets received. When in this mode, each graph contains two lines – one showing the individual processor node's packets received, and, for comparison, one showing the aggregate number of packets received. To distinguish between the two, the area between the individual packets received line and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on the line representing either aggregate or individual packets received.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate packets received:	If the cursor is on a line representing an individual node's packets received:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of packets received at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of packets received at that point in time. • the individual processor node's number of packets received at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Show Statistics**

▲ The Statistics Window opens. This window is shown on page 138.

Packets Sent (Bar Chart)

This view uses a bar chart to visualize the number of TCP/IP packets sent by processor nodes. Each node is represented as a bar on this chart. If you are using this view for trace visualization, these are the processor nodes that ran your program. If you are using this view for performance monitoring, these are the selected nodes. The length of the bars change to show the number of packets sent. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous number of packets sent. The hatched fill pattern is the average number of packets sent. In addition, VT draws a vertical line to the right of each bar to show the highest instantaneous number of packets sent so far by each processor node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous number of packets sent
- average number of packets sent
- maximum instantaneous number of packets so far sent

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents packets sent as a percentage of 100. You can adjust this so that the view represents packets sent as a percentage of some other value. For example, say the number of packets sent never exceeds 10 percent. You might want to adjust the view so that it represents packets sent as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value.

Packets Sent (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize the number of TCP/IP packets sent by processor nodes. If you are using this view for trace visualization, the processor nodes it depicts are the ones that ran your program. If you are using this view for performance monitoring, these are the selected nodes.

This view has two display modes – aggregate and individual. By default, the view is in aggregate display mode – it uses a single strip graph to represent the average number of packets sent across all processor nodes. Individual display mode shows a separate graph for each processor node's packets sent. When in this mode, each graph contains two lines – one showing the individual processor node's packets sent, and, for comparison, one showing the aggregate number of packets sent. To distinguish between the two, the area between the individual packets sent line and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on the line representing either aggregate or individual packets sent.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate packets sent:	If the cursor is on a line representing an individual node's packets sent:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of packets sent at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of packets sent at that point in time. • the individual processor node's number of packets sent at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Show Statistics**

▲ The Statistics Window opens. This window is shown on page 138.

System Views

The seven views in this category visualize system activities and events. These views are for trace visualization and performance monitoring, and respond to AIX kernel statistic trace records. For more information on AIX kernel statistic trace records, refer to “Types of Trace Records” on page 105.

Context Switches (Bar Chart)

This view uses a bar chart to visualize context switches – the number of times the processes are swapped in and out of active execution. Each node is represented as a bar on the chart. If you are using this view for trace visualization, these are the processor nodes that ran your program. If you are using this view for performance monitoring, these are the selected nodes. The length of the bars change to show the number of context switches. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous number of context switches. The hatched fill pattern is the average number of context switches. In addition, VT draws a vertical line to the right of each bar to show the highest instantaneous number of context switches so far reached for each processor node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous number of context switches
- average number of context switches
- maximum number of context switches so far recorded

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents the number of context switches as a percentage of 100. You can adjust this so that the view represents context switches as a percentage of some other value. For example, say the number of context switches never exceeds 10 percent. You might want to adjust the view so that it represents context switches as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value.

Context Switches (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize context switches – the number of times processes are swapped in and out of active execution. If you are using this view for trace visualization, the processor nodes depicted in this view are the ones that ran your program. If you are using this view for performance monitoring, these are the selected nodes.

This view has two display modes – aggregate and individual. By default, the view is in aggregate display mode – it uses a single strip graph to represent the average number of context switches across all processor nodes. Individual display mode shows a separate graph for each processor node's context switches. When in this mode, each graph contains two lines – one showing the individual processor node's context switches, and, for comparison, one showing the aggregate number of context switches. To distinguish between the two, the area between the line for the individual node and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on the line representing either aggregate or individual context switches.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate context switches:	If the cursor is on a line representing an individual node's context switches:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of context switches at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of context switches at that point in time. • the individual processor node's number of context switches at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Show Statistics**

▲ The Statistics Window opens. This window is shown on page 138.

Page Faults (Bar Chart)

This view uses a bar chart to visualize page faults. Each node is represented as a bar on this chart. If you are using this view for trace visualization, these are the processor nodes that ran your program. If you are using this view for performance monitoring, these are the selected nodes. The length of the bars change to show the number of page faults. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous number of page faults. The hatched fill pattern is the average number of page faults. In addition, VT draws a vertical line to the right of each bar to show the highest instantaneous number of page faults so far by each processor node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous number of page faults
- average number of page faults
- maximum number of page faults so far recorded

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents page faults as a percentage of 100. You can adjust this so that the view represents page faults as a percentage of some other value. For example, say the number of page faults never exceeds 10 percent. You might want to adjust the view so that it represents page faults as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value.

Page Faults (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize page faults. If you are using this view for trace visualization, the processor nodes it depicts are the ones that ran your program. If you are using this view for performance monitoring, these are the selected nodes.

This view has two display modes – aggregate and individual. By default, the view is in aggregate display mode – it uses a single strip graph to represent the average number of page faults across all processor nodes. Individual display mode shows a separate graph for each processor node's page faults. When in this mode, each graph contains two lines – one showing the individual processor node's page faults, and, for comparison, one showing the aggregate number of page faults. To distinguish between the two, the area between the individual page faults line and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on the line representing either aggregate or individual page faults.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate page faults:	If the cursor is on a line representing an individual node's page faults:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of page faults at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of page faults at that point in time. • the individual processor node's number of page faults at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Show Statistics**

▲ The Statistics Window opens. This window is shown on page 138.

System Calls (Bar Chart)

This view uses a bar chart to visualize the number of times processor nodes invoke kernel subroutines. Each node is represented as a bar on this chart. If you are using this view for trace visualization, these are the processor nodes that ran your program. If you are using this view for performance monitoring, these are the selected nodes. The length of the bars change to show the number of system calls. Each bar has both a solid and a hatched fill pattern. The solid fill pattern represents the instantaneous number of system calls. The hatched fill pattern is the average number of system calls. In addition, VT draws a vertical line to the right of each bar to show the highest number of system calls so far reached by each processor node. This view is similar in appearance to the Kernel Utilization (Bar Chart) view shown on page 136. This is an instantaneous view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any of the bars representing processor nodes.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- instantaneous number of system calls
- average number of system calls
- maximum instantaneous number of system calls so far reached

The window remains open only while you hold the mouse button down.

Adjusting the Default Maximum Value: By default, this view represents the number of system calls as a percentage of 100. You can adjust this so that the view represents system calls as a percentage of some other value. For example, say the number of system calls never exceeds 10 percent. You might want to adjust the view so that it represents system calls as a percentage of 10 instead of 100. To do this:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

SELECT **Parameters**

▲ The view's Parameters window opens.

FOCUS on the barMaxValue text entry field.

TYPE IN 10

PRESS **Apply**

▲ The view calibrates to the new maximum value.

System Calls (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize the number of times processor nodes invoke kernel subroutines. If you are using this view for trace visualization, the processor nodes it depicts are the ones that ran your program. If you are using this view for performance monitoring, these are the selected nodes.

This view has two display modes – aggregate and individual. By default, the view is in aggregate display mode – it uses a single strip graph to represent the average number of system calls across all processor nodes. Individual display mode shows a separate graph for each processor node's system calls. When in this mode, each graph contains two lines – one showing the individual processor node's system calls, and, for comparison, one showing the aggregate number of system calls. To distinguish between the two, the area between the individual system calls line and the graph's horizontal axis has a solid fill. This view is similar in appearance to the Kernel Utilization (Graph) view shown on page 137. This is a streaming view.

Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over a point on the line representing either aggregate or individual system calls.

PRESS the left mouse button.

▲ A window opens displaying:

If the cursor is on a line representing aggregate system calls:	If the cursor is on a line representing an individual node's system calls:
• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of system calls at that point in time.	• time index. This is the time during the program's execution that is currently being depicted in the view. • the aggregate number of system calls at that point in time. • the individual processor node's number of system calls at that point in time.

The window remains open only while you hold the mouse button down.

Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Individual**

Displaying an Analysis of the Graph Information: You can open a Statistics Window containing an analysis of the information – the mean and the standard deviation – shown in the view. To open the Statistics Window:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Show Statistics**

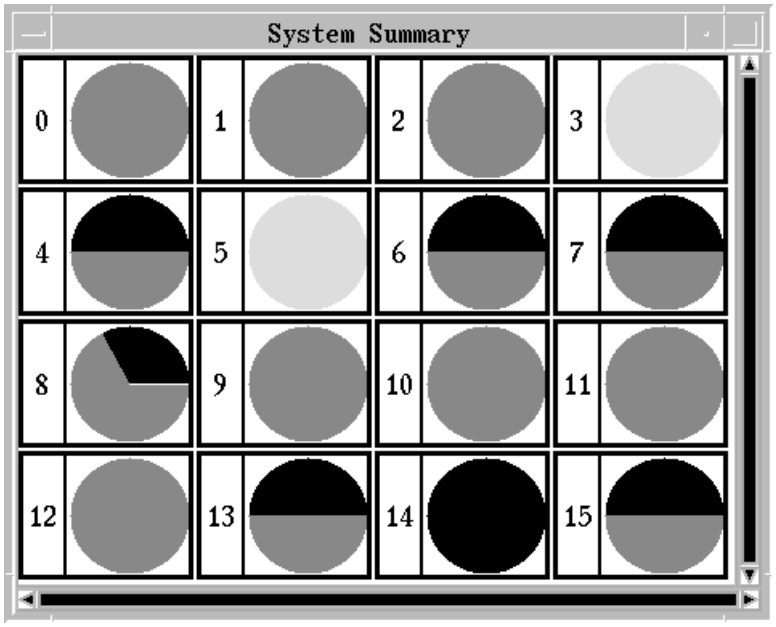
▲ The Statistics Window opens. This window is shown on page 138.

System Summary

This view uses a series of pie charts to summarize system activity. Each pie chart represents a processor node, and is segmented to show the percentage of time the node spent:

- idle
- executing a program
- executing a kernel function
- waiting for some resource to become available. For example, waiting for a disk to become available.

If you are using this view for trace visualization, the processor nodes shown are the ones that ran your program. If you are using this view for performance monitoring, these are the selected nodes. This is an instantaneous view.



Displaying Additional Data in the View: You may display additional data in this view. To do this:

PLACE the mouse cursor over any pie chart.

PRESS the left mouse button.

▲ A window opens displaying the:

- time index. This is the time during the program's execution that is currently being depicted in the view.
- node number
- system state represented. This includes:
  - user time

  - kernel time
  - wait time
  - idle time
- percent of time spent in that state

Toggling Between Instantaneous and Cumulative Presentation: The view lets you toggle between instantaneous and cumulative presentation of system activity. By default, the pie charts summarize instantaneous system activity. A cumulative presentation shows overall system activity for each processor node.

If you want a cumulative system summary:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

▲ A selection menu appears.

SELECT **Cumulative**

Customizing the View's Colors: As described in “Adjusting a View's Display Rate and Colors” on page 101, the appearance of a view – the colors it uses – is determined by its display spectrum. The CPU Load display spectrum was designed specifically for use with this view. Each color in this spectrum is a resource you can customize using the *.Xdefaults* file. This enables you to specify the exact color used to represent the four system states in the view. For more information, see “VT Resource Variables” on page 220

Appendix A. Parallel Environment Commands

This appendix contains the manual pages for the PE commands discussed throughout this book. Each manual page is organized into sections. The sections always appear in the same order, but some appear in all manual pages while others are optional. The manual page sections are:

NAME Provides the name of the command described in the manual page, and a brief description of its purpose.

SYNOPSIS Includes a diagram that summarizes the command syntax, and provides a brief synopsis of its use and function. If you are unfamiliar with the typographic conventions used in the syntax diagrams, see “Document Conventions” on page viii.

FLAGS Lists and describes any required and optional flags for the command.

DESCRIPTION Describes the command more fully than the **NAME** and **SYNOPSIS** sections.

ENVIRONMENT VARIABLES
Lists and describes any applicable environment variables.

EXAMPLES Provides examples of ways in which the command is typically used.

FILES Lists and describes any files related to the command.

RELATED INFORMATION
Lists commands, functions, file formats, and special files that are employed by the command, that have a purpose related to the command, or that are otherwise of interest within the context of the command.

© Copyright IBM Corp. 1993

mpcc

NAME

mpcc – Invokes a shell script to compile C programs.

SYNOPSIS

mpcc [*cc_flags*]... *program.c* [**-ip** | **-us**]

The **mpcc** shell script compiles C programs while linking in the Partition Manager and message-passing interface libraries. You can optionally link in either the IP or the US CSS library implementation.

FLAGS

-ip Statically binds the IP CSS library implementation with the executable. This CSS library implementation uses Internet Protocol for communication among processor nodes. If the POWERparallel systems' high-performance switch feature is not available, the IP CSS must be used for communication. The IP CSS can be statically linked now, or else dynamically linked when the executable is invoked.

-us Statically binds the US CSS library implementation with the executable. This CSS library implementation uses the US protocol for dedicated use of the POWERparallel system's high-performance switch adapter. It allows the parallel tasks of the program to drive the switch adapter directly. The US CSS requires a POWERparallel system configured with the high-performance switch feature. The US CSS can be statically linked now, or else dynamically linked when the executable is invoked.

Any of the compiler flags normally accepted by the **cc** command can also be used on **mpcc**. For a complete listing of these flag options, refer to the manual page for the AIX **cc** command. Typical options to **mpcc** include:

-v causes a “verbose” output listing of the shell script.

-g Produces an object file with symbol table references. This object file is needed for debugging with the **pdbx** and **xpdbx** debuggers, and is also needed by the Source Code view of the Visualization Tool (**vt**).

-o names the executable.

-l (lower-case L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the FILES section.

-I (upper-case i)
names directories for additional includes. The directory */usr/lpp/poe/include* is automatically included.

-p enables profiling with the **prof** command. When specifying this option, you need to set the environment variable **MP_LIBPATH** to */usr/lib/profiled* to get current information for system resources.

-pg enables profiling with the **gprof** command. When specifying this option, you need to set the environment variable **MP_LIBPATH** to */usr/lib/profiled* to get current information for system resources.

DESCRIPTION

The **mpcc** shell script calls another shell script (**mputil**), which in turn calls the AIX **cc** compiler. In addition, the Partition Manager and message-passing interface libraries are automatically linked in.

Flags are passed by **mpcc** to the **cc** command, so any of the **cc** options can be used on the **mpcc** shell script. In addition, the **-ip** and the **-us** flags can be used to bind either the IP or the US implementation of the communication subsystem (CSS) library with the executable. If neither flag is used, the CSS library implementation can be dynamically linked when you invoke the executable using the **poe** command. The value specified by the **MP_EUILIB** environment variable or the **-euilib** flag will then determine which CSS library implementation is dynamically linked.

ENVIRONMENT VARIABLES

MP_PREFIX sets an alternate path to the scripts library. If not set or NULL, the standard path */usr/lpp/poe* is used. If this environment variable is set, then all libraries are prefixed by *$MP_PREFIX/poe*.

EXAMPLES

1. To compile a C program to statically bind the IP CSS library implementation with the executable, enter:

   ```
   mpcc program.c -ip -o program
   ```

2. To compile a C program to statically bind in the US CSS library implementation with the executable, enter:

   ```
   mpcc program.c -us -o program
   ```

3. To compile a C program without binding either CSS library implementation with the executable, enter:

   ```
   mpcc program.c -o program
   ```

FILES

When you compile a program using **mpcc**, the following libraries are automatically selected:

/usr/lpp/poe/lib/libeui1c.a (Message-passing interface, parallel utilities)

/usr/lpp/poe/lib/libccl.a (Collective communication library)

/usr/lib/libs.a (System library)

/usr/lib/libm.a (Math library)

/usr/lpp/poe/lib/libvtd.a (VT tracing subsystem)

/usr/lpp/poe/lib/libppe.a (PE common routines)

If the **-ip** option flag is specified, the following library is also included:

/usr/lpp/poe/lib/ip/libcss.a (IP CSS library implementation)

If the **-us** option flag is specified, the following library is also included:

/usr/lpp/poe/lib/us/libcss.a (US CSS library implementation)

/usr/lpp/ssp/css/libus/fs_ext.exp (Resolves exported symbols)

RELATED INFORMATION

Commands: **mpCC**(1), **mpxlf**(1), **cc**(1), **pdbx**(1), **xpdbx**(1), **vt**(1)

mpCC

NAME

mpCC – Invokes a shell script to compile C++ programs.

SYNOPSIS

mpCC [*xlC_flags*]... *program.C* [**-ip** | **-us**]

The **mpCC** shell script compiles C++ programs while linking in the Partition Manager and message-passing interface libraries. You can optionally link in either the IP or the US CSS library implementation.

FLAGS

-ip Statically binds the IP CSS library implementation with the executable. This CSS library implementation uses Internet Protocol for communication among processor nodes. If the POWERparallel systems' high-performance switch feature is not available, the IP CSS must be used for communication. The IP CSS can be statically linked now, or else dynamically linked when the executable is invoked.

-us Statically binds the US CSS library implementation with the executable. This CSS library implementation uses the US protocol for dedicated use of the POWERparallel system's high-performance switch adapter. It allows the parallel tasks of the program to drive the switch adapter directly. The US CSS requires a POWERparallel system configured with the high-performance switch feature. The US CSS can be statically linked now, or else dynamically linked when the executable is invoked.

Any of the compiler flags normally accepted by the **xlC** command can also be used on **mpCC**. For a complete listing of these flag options, refer to the manual page for the **xlC** command. Typical options to **mpCC** include:

-v causes a “verbose” output listing of the shell script.

-g Produces an object file with symbol table references. This object file is needed for debugging with the **pdbx** and **xpdbx** debuggers, and is also needed by the Source Code view of the Visualization Tool (**vt**).

-o names the executable.

-l (lower-case L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the FILES section.

-I (upper-case i)
names directories for additional includes. The directory */usr/lpp/poe/include* is automatically included.

-p enables profiling with the **prof** command. When specifying this option, you need to set the environment variable **MP_LIBPATH** to */usr/lib/profiled* to get current information for system routines.

-pg enables profiling with the **gprof** command. When specifying this option, you need to set the environment variable **MP_LIBPATH** to */usr/lib/profiled* to get current information for system routines.

DESCRIPTION

The **mpCC** shell script calls another shell script (**mputil**) which in turn calls the AIX **xlC** compiler. In addition, the Partition Manager and message-passing interface libraries are automatically linked in.

Flags are passed by **mpCC** to the **xlC** command, so any of the **xlC** options can be used on the **mpCC** shell script. In addition, the **-ip** and the **-us** flags can be used to bind either the IP or the US implementation of the communication subsystem (CSS) library with the executable. If neither flag is used, the CSS library implementation can be dynamically linked when you invoke the executable using the **poe** command. The value specified by the **MP_EUILIB** environment variable or the **-euilib** flag will then determine which CSS library implementation is dynamically linked.

ENVIRONMENT VARIABLES

MP_PREFIX sets an alternate path to the scripts library. If not set or NULL, the standard path */usr/lpp/poe* is used. If this environment variable is set, then all libraries are prefixed by *$MP_PREFIX/poe*.

EXAMPLES

1. To compile a C++ program to statically bind the IP CSS library implementation with the executable, enter:

   ```
   mpCC program.C -ip -o program
   ```

2. To compile a C++ program to statically bind in the US CSS library implementation with the executable, enter:

   ```
   mpCC program.C -us -o program
   ```

3. To compile a C++ program without binding either CSS library implementation with the executable, enter:

   ```
   mpCC program.C -o program
   ```

FILES

When you compile a program using **mpCC**, the following libraries are automatically selected:

/usr/lpp/poe/lib/libeui1c.a (Message-passing interface, parallel utilities)

/usr/lpp/poe/lib/libccl.a (Collective communication library)

/usr/lib/libs.a (System library)

/usr/lib/libm.a (Math library)

/usr/lpp/poe/lib/libvtd.a (VT tracing subsystem)

/usr/lpp/poe/lib/libppe.a (PE common routines)

If the **-ip** option flag is specified, the following library is also included:

/usr/lpp/poe/lib/ip/libcss.a (IP CSS library implementation)

If the **-us** option flag is specified, the following library is also included:

/usr/lpp/poe/lib/us/libcss.a (US CSS library implementation)

/usr/lpp/ssp/css/libus/fs_ext.exp (Resolves exported symbols)

RELATED INFORMATION

Commands: **mpcc**(1), **mpxlf**(1), **xlC**(1), **pdbx**(1), **xpdbx**(1), **vt**(1)

mprcp

NAME

mprcp – copies a file from the Home Node to a list of remote hosts.

SYNOPSIS

mprcp *host_list_file file_id*

The **mprcp** shell script uses the **rcp** command to copy the file specified by *file_id* to all the remote hosts listed in the specified *host_list_file*. This *file_id* must be specified by an absolute path name.

FLAGS

None.

DESCRIPTION

The **mprcp** shell script is typically used to distribute executables, data files, and (in order to use the parallel debuggers) source code files from the Home Node to all the remote nodes of the partition prior to invoking **poe**. This only needs to be done if the needed files are not in a shared file system, or are not part of a file collection which is distributed automatically.

For each remote host listed in the specified *host_list_file*, the size and date of *file_id* on the remote system is determined. If the file does not exist on the remote system, or if either the size or date differs from the corresponding statistic for the local system, the **rcp** command is used to copy the file to the remote system. The copied file retains the local system size and date and overlays any existing file of the same name. The remote copy uses the same userid as the local system. If the remote host cannot be contacted, or if the **rcp** command fails, an error message is printed and the script exits.

EXAMPLES

To send a copy of the executable *program* to all the processor nodes listed in *host.list* in your current directory, enter:

```
mprcp host.list $PWD/program
```

RELATED INFORMATION

Commands: **rcp**(1)

mpxlf

NAME

mpxlf – Invokes a shell script to compile FORTRAN programs.

SYNOPSIS

mpxlf [*xlf_flags*]... *program.f* [**-ip** | **-us**]

The **mpxlf** shell script compiles FORTRAN programs while linking in the Partition Manager and message-passing interface libraries. You can optionally link in either the IP or the US CSS library implementation.

FLAGS

-ip Statically binds the IP CSS library implementation with the executable. This CSS library implementation uses Internet Protocol for communication among processor nodes. If the POWERparallel systems' high-performance switch feature is not available, the IP CSS must be used for communication. The IP CSS can be statically linked now, or else dynamically linked when the executable is invoked.

-us Statically binds the US CSS library implementation with the executable. This CSS library implementation uses the US protocol for dedicated use of the POWERparallel system's high-performance switch adapter. It allows the parallel tasks of the program to drive the switch adapter directly. The US CSS requires a POWERparallel system configured with the high-performance switch feature. The US CSS can be statically linked now, or else dynamically linked when the executable is invoked.

Any of the compiler flags normally accepted by the **xlf** command can also be used on **mpxlf**. For a complete listing of these flag options, refer to the manual page for the **xlf** command. Typical options to **mpxlf** include:

-v causes a “verbose” output listing of the shell script.

-g Produces an object file with symbol table references. This object file is needed for debugging with the **pdbx** and **xpdbx** debuggers, and is also needed by the Source Code view of the Visualization Tool.

-o names the executable.

-l (lower-case L)
names additional libraries to be searched. Several libraries are automatically included, and are listed below in the FILES section.

-I (upper-case i)
names directories for additional includes. The directory */usr/lpp/poe/include* is automatically included.

-p enables profiling with the **prof** command. When specifying this option, you need to set the environment variable **MP_LIBPATH** to */usr/lib/profiled* to get current information for system routines.

-pg enables profiling with the **gprof** command. When specifying this option, you need to set the environment variable **MP_LIBPATH** to */usr/lib/profiled* to get current information for system routines.

DESCRIPTION

The **mpxlf** shell script calls another shell script (**mputil**), which in turn calls the **xlf** compiler. In addition, the Partition Manager and message-passing interface libraries are automatically linked in.

Flags are passed by **mpxlf** to the **xlf** command, so any of the **xlf** options can be used on the **mpxlf** shell script. In addition, the **-ip** and the **-us** flags can be used to bind either the IP or the US implementation of the communication subsystem (CSS) library with the executable. If neither flag is used, the CSS library implementation can be dynamically linked when you invoke the executable using the **poe** command. The value specified by the **MP_EUILIB** environment variable or the **-euilib** flag will then determine which CSS library implementation is dynamically linked.

ENVIRONMENT VARIABLES

MP_PREFIX sets an alternate path to the scripts library. If not set or NULL, the standard path */usr/lpp/poe* is used. If this environment variable is set, then all libraries are prefixed by *$MP_PREFIX/poe*.

EXAMPLES

1. To compile a FORTRAN program to statically bind the IP CSS library implementation with the executable, enter:

   ```
   mpxlf program.f -ip -o program
   ```

2. To compile a FORTRAN program to statically bind in the US CSS library implementation with the executable, enter:

   ```
   mpxlf program.f -us -o program
   ```

3. To compile a FORTRAN program without binding either CSS library implementation with the executable, enter:

   ```
   mpxlf program.f -o program
   ```

FILES

When you compile a program using **mpxlf**, the following libraries are automatically selected:

/usr/lpp/poe/lib/libeui1f.a (Message-passing interface, parallel utilities)

/usr/lpp/poe/lib/libccl.a (Collective communication library)

/usr/lib/libs.a (System library)

/usr/lib/libm.a (Math library)

/usr/lpp/poe/lib/libvtd.a (VT tracing subsystem)

/usr/lpp/poe/lib/libppe.a (PE common routines)

If the **-ip** option flag is specified, the following library is also included:

/usr/lpp/poe/lib/ip/libcss.a (IP CSS library implementation)

If the **-us** option flag is specified, the following library is also included:

/usr/lpp/poe/lib/us/libcss.a (US CSS library implementation)

/usr/lpp/ssp/css/libus/fs_ext.exp (Resolves exported symbols)

RELATED INFORMATION

Commands: **mpcc**(1), **mpCC**(1), **xlf**(1), **pdbx**(1), **xpdbx**(1), **vt**(1).

pdbx

NAME

pdbx – Invokes the **pdbx** debugger, which is the command-line interface of the PE debugging facility.

SYNOPSIS

pdbx [*program*] [*program_options*] [*POE_options*] [**-c** *commands_file*] [**-f**] [**-I** *directory*]... [**-d** *nesting_depth*] [**-t** *table_size*] [**-x**]

The **pdbx** command invokes the **pdbx** debugger, which is the command-line interface of the PE debugging facility. This facility is based on the **dbx** debugger, but adds function specific to parallel programming.

FLAGS

Because the **pdbx** command runs in the Parallel Operating Environment, it accepts all the flags supported by the **poe** command. See the **poe** manual page for a description of these options. Additional **pdbx** flags are:

-c Reads startup commands from the specified *commands_file*.

-f Reads only required symbol table information upon initiation of the **dbx** sessions. You can use this flag to speed startup and to cut down memory usage. This is especially useful when debugging large executable files and when page space is low.

-I (upper-case i)
Adds the specified *directory* to the list of directories searched for an executable's source files. (Once **pdbx** is running, this list can also be overridden on a group or single node basis with the **use** subcommand.)

-d Sets the limit for the nesting of program blocks. The default nesting depth limit is 25.

-t Increases the size of each **dbx**'s type table by the *table_size* number variable. The **-t** flag is needed when an executable file contains more than 5000 types.

-x Prevents the **dbx** command from stripping _ (trailing underscore) characters from symbols originating in FORTRAN source code. This flag enables **dbx** to distinguish between symbols which are identical except for an underscore character, such as xxx and xxx_.

-tmpdir This *POE_option* flag is normally associated with Visualization Tool trace collection. It specifies the directory to which output trace files are written. For **pdbx**, it specifies the directory to which the individual startup files (.pdbxinit.*process_id.task_id*) are written for each **dbx** task. This is frequently local, and may be a shared directory. If not set, and if its associated environment variable **MP_TMPDIR** is not set, the default location is */tmp*.

DESCRIPTION

The **pdbx** command is the Parallel Environment's debugging facility for parallel programs. It is based, and built, on the AIX debugging tool **dbx**.

The **pdbx** debugger supports most of the familiar **dbx** subcommands, as well as additional **pdbx** subcommands.

To use **pdbx** for interactive debugging you first need to compile the program and set up the execution environment as you would to invoke a parallel program with the **poe** command. Your program should be compiled with the **-g** flag in order to produce an object file with symbol table references.

The **pdbx** debugger maintains the **dbx** debugger's command-line interface and subcommands. When you issue the **pdbx** command, the **pdbx** command prompt displays to mark the start of a **pdbx** session. The **pdbx** debugger supports most of the familiar **dbx** subcommands, as well as additional **pdbx** subcommands.

When using the **pdbx** debugger, you should keep in mind that **pdbx** subcommands can either be context sensitive or context insensitive. In **pdbx**, context refers to a setting that controls which task(s) receive the subcommands entered at the **pdbx** command prompt. A default command context is provided which contains all tasks in your partition. You can, however, set the command context on a single task or a group of tasks. Context sensitive subcommands, when entered, only effect those tasks in the current command context. Context insensitive subcommands are not effected by the command context setting.

If you are already familiar with **dbx**, you should be aware that some of the **dbx** subcommands behave somewhat differently in **pdbx**. Be aware that:

- all the **dbx** subcommands are context sensitive in **pdbx**. If you use the **stop** subcommand, for example, it will only set breakpoints for the tasks in the current context. Tasks outside the current context are not affected.
- redirection from **dbx** subcommands is not supported.
- you cannot use the subcommands **clear**, **detach**, **edit**, **multproc**, **prompt**, **run**, **rerun**, and **screen**.
- since **pdbx** runs in the Parallel Operating Environment, output from the parallel tasks may not be ordered. You can force task ordering, however, by setting the output mode to *ordered* using the **MP_STDOUTMODE** environment variable or the **-stdoutmode** flag when invoking your program with **pdbx**.

ENVIRONMENT VARIABLES

Because the **pdbx** command runs in the Parallel Operating Environment, in interacts with the same environment variables associated with the **poe** command. See the **poe** manual page for a description of these environment variables. Additional variables are:

HOME During **pdbx** initialization, **pdbx** uses this environment variable to search for two special initialization files. First, **pdbx** searches for *$HOME/.pdbxinit*, and then *./pdbxinit*.

SHELL The **sh** subcommand in **dbx**, which is available through **pdbx**, uses this environment variable to determine which shell to use. If this environment variable is not set, the default is the **sh** shell.

MP_DBXPROMPTMOD
The **dbx** prompt *\n(dbx)* is used by **pdbx** as an indicator denoting that a **dbx** subcommand has completed. This environment variable can be used to modify the prompt. Any value assigned to **MP_DBXPROMPTMOD** will have a "." prepended and then be inserted in the *\n(dbx)* prompt between the "x" and the ")". This environment variable is useful when the string *\n(dbx)* is present in the output of the application being debugged. For example, if **MP_DBXPROMPTMOD** is set to *unique157*, the prompt would be *\n(dbx.unique157)*.

MP_TMPDIR This environment variable is normally associated with Visualization Tool trace collection. In trace collection, it specifies the directory to which output trace files are written. For **pdbx**, it specifies the directory to which the individual startup files (.pdbxinit.*process_id.task_id*) are written for each **dbx** task. This is frequently local, and may be a shared directory. If not set, and if its associated command-line flag **tmpdir** is not used, the default location is */tmp*.

EXAMPLES

To start the **pdbx** debugger, first set up the execution environment as you would for the **poe** command, and then enter:

```
pdbx
```

You should be presented with the prompt:

```
pdbx(all)
```

FILES

.pdbxinit (Initial commands for **pdbx**)

.pdbxinit.*process_id.task_id* (Initial commands for the individual **dbx** tasks)

RELATED INFORMATION

Commands: **dbx**(1), **mpcc**(1), **mpCC**(1), **mpxlf**(1), **xpdbx**(1), **poe**(1)

"Debugging Programs" in *AIX for RISC System/6000: General Programming Concepts*, (SC23-2205).

SUBCOMMANDS

active Subcommand for the **pdbx** Command

Lists the tasks that are active (or loaded) in the partition.

SYNOPSIS

active

DESCRIPTION

The **active** subcommand lists the task numbers of all the tasks in the partition that are in active state. In other words, all the **dbx** tasks that have an application loaded and ready to debug. The active subcommand is context insensitive, and has no parameters.

EXAMPLES

Consider an application with 5 tasks available. You have defined a task group consisting of tasks 0, 2, and 4. By setting the context on this group and using the **load** subcommand, you have loaded a program for debugging on the **dbx** tasks 0, 2, and 4. To list the active tasks, enter:

```
active
```

The **pdbx** debugger responds with:

```
  0    2    4
```

Similar information is available using the **list** action of the **group** subcommand.

```
group list
```

The **pdbx** debugger responds with:

```
all                                0 A    1      2 A    3      4 A
partial                            0 A    2 A    4 A
```

RELATED INFORMATION

The **group** subcommand for the **pdbx** command.

---

alias Subcommand for the **pdbx** Command

Creates aliases for **pdbx** subcommands.

SYNOPSIS

alias [*alias_name* [*alias_string*]]

DESCRIPTION

The **alias** subcommand creates aliases for **pdbx** subcommands. The *alias_name* parameter is the alias being created. The *alias_string* is the **pdbx** subcommand for which you wish you define an alias, and is a single **pdbx** subcommand. If used without parameters, the **alias** subcommand displays all current aliases. If only *alias_name* is specified, it lists the alias name and the alias string that is assigned to it. This subcommand is context insensitive.

A number of default aliases are provided by **pdbx**. They are:

t	where
j	status
st	stop
s	step
x	registers
q	quit
p	print
n	next
m	map
l	list
h	help
d	delete
c	cont

Apart from these, aliases are only known during the current **pdbx** session. They are not saved between **pdbx** sessions, and are lost upon exiting **pdbx**.

Note: One method for reusing aliases is to define them in *.pdbxinit* to allow them to be created for each **pdbx** execution. The default aliases are available after the partition has been loaded.

Aliases can also be removed using the **unalias** subcommand for the **pdbx** command.

EXAMPLES

1. If you have two task groups defined in your **pdbx** session called “master” and “slaves”, and you wish to define aliases to easily qualify each, enter:

   ```
   alias mas on master
   alias sla on slaves
   ```

   This will allow you to switch the command context between the master and slaves groups by typing:

   ```
   mas
   ```

   to switch context to the “master” group, or:

   ```
   sla
   ```

   to switch context to the “slaves” group.

2. To display the string that has been defined for the alias “p”, enter:

   ```
   alias p
   ```

3. To list all aliases currently defined, enter:

   ```
   alias
   ```

RELATED INFORMATION

The **unalias** subcommand of the **pdbx** command.

dbx Subcommand for the **pdbx** Command.

Issues a **dbx** subcommand directly to **dbx**.

SYNOPSIS

dbx *dbx_subcommand*

DESCRIPTION

The **dbx** subcommand is context sensitive and will pass the specified *dbx_subcommand* directly to the **dbx** running on each task in the current context with no **pdbx** intervention. The specified *dbx_subcommand* can be any valid **dbx** subcommand.

Note: The **pdbx** command uses **dbx** to access tasks on individual nodes. In many cases, **pdbx** saves and requires its own state information about the tasks. Some **dbx** commands will circumvent the ability of **pdbx** to maintain accurate state information about the tasks being debugged. Therefore, use the **dbx** subcommand with caution. In general, **dbx** subcommands used to display information will have no adverse side effects. The dbx subcommands **clear**, **detach**, **edit**, **multproc**, **prompt**, **run**, **rerun**, and **screen** are currently unsupported under **pdbx** and should not be used.

EXAMPLES

To display the events that the **dbx** running as task 1 recognizes, enter:

```
on 1 dbx status
```

RELATED INFORMATION

Commands: **dbx**(1)

---

delete Subcommand for the **pdbx** Command

Deletes **pdbx** breakpoints and tracepoints within a given context.

SYNOPSIS

delete *event_ID*

DESCRIPTION

The **delete** subcommand can be used to remove **pdbx** breakpoints and tracepoints. The *event_ID* is the number provided by the **status** subcommand.

EXAMPLES

Assume the command context is set on task 1 and the output of the **status** subcommand is:

```
1:[0] stop in celsius
all:[0] stop at "foo.c":19
all:[1] trace "foo.c":21
```

To delete all these events, enter the following subcommands:

```
on 1
delete 0
on all
delete 0
delete 1
```

RELATED INFORMATION

The **status** subcommand for the **pdbx** command.
The **stop** subcommand for the **pdbx** command.
The **trace** subcommand for the **pdbx** command.

---

dhelp Subcommand for the **pdbx** Command

Prints out a synopsis of common **dbx** subcommands.

SYNOPSIS

dhelp

DESCRIPTION

The **dhelp** subcommand prints out a brief description of the common **dbx** subcommands. Since **pdbx** supports the majority of these subcommands, the **dhelp** subcommand provides a quick reminder of some of the basic commands available. The **dhelp** subcommand is context insensitive, and has no parameters.

EXAMPLES

To obtain a synopsis of common **dbx** subcommands, enter:

```
dhelp
```

The **pdbx** debugger responds with the list of commonly used **dbx** subcommands. This list is the same as that displayed by the **help** subcommand in regular **dbx**.

RELATED INFORMATION

The **help** subcommand for the **dbx** command.
The **help** subcommand for the **pdbx** command.

---

group Subcommand for the **pdbx** Command

Groups individual tasks under a common name for easier manipulation.

SYNOPSIS

group add *group_name task_list*

group delete *group_name* [task_list]

group change *old_group_name new_group_name*

group list [*group_name*]

Where:
task_list → *task_id* | *task_id*, *task_id* [*task_list*]

DESCRIPTION

The **group** subcommand groups individual tasks under a common name for easier manipulation. It can add or delete a group, add or delete tasks from a group, change the name of a group, list the tasks in a group, or list all groups. This subcommand is context insensitive.

The **add** action adds one or more tasks to a new or existing task group. The *task_list* specified is a list of task identifiers to be included in the new or existing group. These task identifiers can be specified individually in a space-separated list, or as a range using a pair of task identifiers separated by a comma. Individual task identifiers and ranges can also be combined in creating the desired *task_list*.

The **delete** action deletes an existing task group, or deletes one or more tasks from an existing task group. The *task_list*, if specified, is a list of task identifiers to be included in the new or existing group. These task identifiers can be specified individually in a space-separated list, or as a range using a pair of task identifiers separated by a comma. Individual task identifiers and ranges can also be combined in creating the desired *task_list*.

The **change** action changes the name of a task group from *old_group_name* to *new_group_name*.

The **list** action displays the task members for the *group_name* specified, or for all task groups. The task identifiers may be followed by a one-letter status indicator.

A indicates an active task.

E indicates a communication error.

U indicates an unhooked task.

If there is no one-letter status indicator, the task is not active.

EXAMPLES

Consider an application running as five tasks numbered 0 through 4.

1. To create a task group “first” containing task 0, enter:

   ```
   group add first 0
   ```

   The **pdbx** debugger responds with:

   ```
   1 task added to group first
   ```

2. To create a task group “rest” containing tasks 1 through 4, enter:

   ```
   group add rest 1,4
   ```

   The **pdbx** debugger responds with:

   ```
   4 tasks added to group rest
   ```

3. To change the name of the default group “all” to “johnny”, enter:

   ```
   group change all johnny
   ```

   The **pdbx** debugger responds with:

   ```
   'johnny' is now the new name for 'all'
   ```

4. To list all of the groups and the tasks they contain, enter:

   ```
   group list
   ```

   The **pdbx** debugger responds with:

   ```
   johnny                          0 A    1 A    2 A    3 A    4A
   first                           0 A
   rest                            1 A    2 A    3 A    4 A
   ```

5. To delete the group “first”, enter:

   ```
   group delete first
   ```

   To delete members 1, 2 and 3 from group “rest”, enter:

   ```
   group delete rest 1 2 3
   ```

   or

   ```
   group delete rest 1,3
   ```

   The **pdbx** debugger responds with:

   ```
   1 deleted from group 'rest'
   2 deleted from group 'rest'
   3 deleted from group 'rest'
   ```

6. To again list all of the groups and the tasks they contain, enter:

   ```
   group list
   ```

   The **pdbx** debugger responds with:

   ```
   johnny                          0 A    1 A    2 A    3 A    4A
   rest                            4 A
   ```

RELATED INFORMATION

The **on** subcommand for the **pdbx** command.

---

help Subcommand for the **pdbx** Command

Prints out a synopsis of **pdbx** subcommands.

SYNOPSIS

help

DESCRIPTION

The **help** subcommand prints out the syntax and a brief description of the **pdbx** subcommands. The output contains subcommands unique to **pdbx** and select **dbx** commands which function differently under **pdbx**. The output provides a quick reminder of **pdbx** subcommands available. This subcommand is context insensitive and has no parameters.

EXAMPLES

To obtain a synopsis of common **pdbx** subcommands, enter:

```
help
```

RELATED INFORMATION

The **dhelp** subcommand for the **pdbx** command.

---

hook Subcommand for the **pdbx** Command

Reestablishes control over previously unhooked tasks.

SYNOPSIS

hook

DESCRIPTION

The **hook** subcommand allows you to reestablish control over all tasks in the current command context that have been unhooked using the **unhook** subcommand. This subcommand is context sensitive.

EXAMPLES

1. To reestablish control over task 2 if it has been unhooked, enter:

   ```
   on 2 hook
   ```

   or

   ```
   on 2
   hook
   ```

2. To reestablish control over all unhooked tasks in the task group “rest”, enter:

   ```
   on rest hook
   ```

   or

   ```
   on rest
   hook
   ```

Listing the members of the task group “all” using the **list** action of the **group** subcommand will allow you to check which tasks are hooked and which are unhooked. Enter:

```
group list all
```

The **pdbx** debugger will display a list similar to the following:

```
0 A    1 U    2 A    3 A
```

Tasks marked with the letter A next to them are active, hooked tasks. In this case, tasks 0, 2, and 3 are active. Tasks marked with the letter U are unhooked. In this case, task 1 is unhooked.

RELATED INFORMATION

The **detach** subcommand for the **dbx** command.
The **unhook** subcommand for the **pdbx** command.

load Subcommand for the **pdbx** Command

Loads a user program for debugging on one or more **dbx** task.

SYNOPSIS

load *program* [*program_options*]

DESCRIPTION

The **load** subcommand loads the specified application *program* to be debugged on the task(s) in the current context. You can optionally specify *program_options* to be passed to the application program. The **load** subcommand is context sensitive. All tasks in the partition must be in the active state (have an application program loaded) before other context sensitive subcommands can be issued. This subcommand enables you to individually or selectively load programs. If you wish to load the same program on all tasks in the partition, the name of the program can be passed as an argument to the **pdbx** command at startup.

EXAMPLES

To load the program "mpprob1" on all tasks in the current context, enter:

```
load mpprob1
```

RELATED INFORMATION

None.

---

on Subcommand for the **pdbx** Command

Sets the current context, or issues a subcommand in an explicit context.

SYNOPSIS

on {*group_name* | *task_id*} [*subcommand*]

DESCRIPTION

The **on** subcommand sets the current command context used to direct subsequent subcommands at a specific task or group of tasks. The context can be set on a task group (by specifying a *group_name*) or on a single task (by specifying a *task_id*).

When a context sensitive *subcommand* is specified, it is directed to the given context without changing the current command context. Thus, specifying the optional *subcommand* enables you to temporarily deviate from the command context.

By using the **on** and **group** subcommands, the number of subcommands issued and the amount of debug data displayed can be tailored to manageable amounts.

EXAMPLES

Assume you have an application running as 15 tasks, and the output of the **group** subcommand lists the existing task groups as:

```
all                                 0 A    1 U    2 A    3 A    4 A    5 A  ...
johnny                              0 A
jessica                             1 U    2 A    3 A    6 U    8 A
unhooked                            1 U    6 U   13 U   14 U
```

1. To add a breakpoint for task 0, enter:

   ```
   on johnny stop at 31
   ```

   The **pdbx** debugger responds with a message like:

   ```
   stop at "ring.f":31
   ```

2. To add breakpoints for all of the tasks in the task group “jessica”, enter:

   ```
   on jessica stop in ring
   ```

   The **pdbx** debugger responds with:

   ```
   stop in ring
   ```

3. To switch the current context to the task group “johnny”, enter:

   ```
   on johnny
   ```

   The **pdbx** debugger responds with the prompt:

   ```
   pdbx(johnny)
   ```

   To add a conditional breakpoint for all tasks in the current context, enter:

   ```
   stop at 48 if len < 1
   ```

   The **pdbx** debugger responds with a message like:

   ```
   stop at "ring.f":48 if len < 1
   ```

4. To view the events that have been set on the task group “jessica”, enter:

   ```
   on jessica status
   ```

   The **pdbx** debugger responds with a message like:

   ```
   jessica:[0] stop in ring
   ```

5. To add a tracepoint for task 1, enter:

   ```
   on 1
   ```

   The **pdbx** debugger responds with the prompt:

   ```
   pdbx(1)
   ```

   Then, enter:

   ```
   trace 57
   ```

   The **pdbx** debugger responds with a message like:

   ```
   trace "ring.f":57
   ```

6. To view the events that have been set for task 1, enter:

   ```
   status
   ```

   The **pdbx** responds with a list like:

   ```
   1:[0] trace "ring.f":57
   jessica:[0] stop in ring
   ```

RELATED INFORMATION

The **group** subcommand for the **pdbx** command.

---

quit Subcommand for the **pdbx** Command

Stops the **pdbx** debug program.

SYNOPSIS

quit

DESCRIPTION

The **quit** subcommand terminates all program tasks, and ends the **pdbx** debugging session. The **quit** subcommand is context insensitive and has no parameters.

EXAMPLES

To exit the **pdbx** debug program, enter:

```
quit
```

RELATED INFORMATION

None.

---

source Subcommand for the **pdbx** Command

Reads **pdbx** subcommands from a specified file.

SYNOPSIS

source *commands_file*

DESCRIPTION

The **source** subcommand reads **pdbx** subcommands from the specified *commands_file*. The *commands_file* should reside on the node where **pdbx** was issued and can contain any commands that are valid on the **pdbx** command line. The **source** subcommand is context insensitive.

EXAMPLES

To read **pdbx** subcommands from a file named “jessica”, enter:

```
source jessica
```

RELATED INFORMATION

None.

status Subcommand for the **pdbx** Command

Displays a list of **pdbx** tracepoints and breakpoints set within tasks in the current context.

SYNOPSIS

status

DESCRIPTION

The **status** subcommand displays a list of **pdbx** tracepoints and breakpoints for tasks in the current context. These tracepoints and breakpoints are sometimes referred to as **pdbx** "events".

Each line of status displayed will have the following format:

```
context: [event-ID] event-description
```

The `event-ID` displayed can later be used by the **delete** subcommand to remove the event. Any events that have been set for the task group "all" will always be displayed. The **status** subcommand is context sensitive.

EXAMPLES

Assume the following commands have been issued, setting various breakpoints and tracepoints.

```
on all
stop at 19
trace 21
on 0
trace foo at 21
on 1
stop in func
```

To display a list of breakpoints and tracepoints for tasks in the current "task 1" context, enter:

```
status
```

The **pdbx** debugger responds with lines of status like:

```
1:[0] stop in func
all:[0] stop at "foo.c":19
all:[1] trace "foo.c":21
```

Notice that the status from the "task 0" context does not get displayed since the context is on "task 1". Also notice that event 0 is unique for the "task 1" context and the "group all" context.

RELATED INFORMATION

The **stop** subcommand for the **pdbx** command.
The **trace** subcommand for the **pdbx** command.
The **delete** subcommand for the **pdbx** command.

---

stop Subcommand for the **pdbx** Command

Sets a breakpoint within tasks in the current context.

SYNOPSIS

stop if *condition*

stop [*variable*] **at** *souceline* [**if** *condition*]

stop [*variable*] **in** *procedure* [**if** *condition*]

stop *variable* [**if** *condition*]

DESCRIPTION

The **stop** subcommand sets stopping places called “breakpoints” for tasks in the current context. Use it to mark these stopping places, and then run the program. When the tasks reach a breakpoint, execution stops and the state of the program can then be examined. The **stop** subcommand is context sensitive.

Use the **status** subcommand to display a list of breakpoints that have been set for tasks in the current context. Use the **delete** subcommand to remove breakpoints.

Notes:

1. The **pdbx** debugger will not attempt to set a breakpoint at a line number when in a group context if the group members (tasks) have different current source files.
2. When specifying variable names as arguments to the **stop** subcommand, fully qualified names should be used. This should be done because, when a **stop** subcommand is issued, a parallel application could be in a different function on each node. This may result in ambiguity in variable name resolution. Use the **which** subcommand to get the fully qualified name for a variable.

EXAMPLES

To set a breakpoint at line 19 of a program, enter:

```
stop at 19
```

The **pdbx** debugger responds with a message like:

```
all:[0] stop at "foo.c":19
```

RELATED INFORMATION

The **stop** subcommand for the **dbx** command.
The **which** subcommand for the **dbx** command.
The **trace** subcommand for the **pdbx** command.
The **status** subcommand for the **pdbx** command.
The **delete** subcommand for the **pdbx** command.

trace Subcommand for the **pdbx** Command

Sets a tracepoint within task in the current context.

SYNOPSIS

trace

trace [*sourceline* | *expression* **at** *sourceline*] [**if** *condition*]

trace *procedure* [**in** *procedure*] [**if** *condition*]

trace [*variable*] [**in** *procedure* | **at** *sourceline*] [**if** *condition*]

DESCRIPTION

The **trace** subcommand sets tracepoints for tasks in the current context. These tracepoints will cause tracing information for the specified *procedure*, *function*, *sourceline*, *expression* or *variable* to be displayed when the program runs. The **trace** subcommand is context sensitive.

Use the **status** subcommand to display a list of tracepoints that have been set in the current context. Use the **delete** subcommand to remove tracepoints.

Notes:

1. The **pdbx** debugger will not attempt to set a tracepoint at a line number when in a group context if the group members (tasks) have different current source files.
2. When specifying variable names as arguments to the **trace** subcommand, fully qualified names should be used. This should be done because, when a **trace** subcommand is issued, a parallel application could be in a different function on each node. This may result in ambiguity in variable name resolution. Use the **which** subcommand to get the fully qualified name for a variable.
3. If the trace generates a large amount of output, it may be helpful to label that output using the **MP_LABELIO** environment variable.

EXAMPLES

To set a tracepoint for the variable "foo" at line 21 of a program, enter:

```
trace foo at 21
```

The **pdbx** debugger responds with a message like:

```
all:[1] trace foo at "bar.c":21
```

RELATED INFORMATION

The **trace** subcommand for the **dbx** command.
The **which** subcommand for the **dbx** command.
The **stop** subcommand for the **pdbx** command.
The **status** subcommand for the **pdbx** command.
The **delete** subcommand for the **pdbx** command.

---

unalias Subcommand for the **pdbx** Command

Removes aliases defined for a **pdbx** session.

SYNOPSIS

unalias *alias_name*

DESCRIPTION

The **unalias** subcommand removes **pdbx** command aliases. The *alias_name* specified is any valid alias that has been defined within your current **pdbx** session. The **unalias** subcommand is context insensitive.

EXAMPLES

To remove the alias “p”, enter:

```
unalias p
```

RELATED INFORMATION

The **alias** subcommand for the **pdbx** command.

---

unhook Subcommand for the **pdbx** Command

Unhooks all tasks in the current command context.

SYNOPSIS

unhook

DESCRIPTION

The **unhook** subcommand enables you to unhook tasks. Unhooking allows tasks to run without intervention from the **pdbx** debugger. You can later reestablish control over unhooked tasks using the **hook** subcommand. The **unhook** subcommand is similar to the **detach** subcommand in **dbx**. It is context sensitive and has no parameters.

EXAMPLES

1. To unhook task 2, enter:

   ```
   on 2 unhook
   ```

   or

   ```
   on 2
   unhook
   ```

2. To unhook all the tasks in the task group “rest”, enter:

   ```
   on rest unhook
   ```

   or

   ```
   on rest
   unhook
   ```

Listing the members of the task group “all” using the **list** action of the **group** subcommand will allow you to check which tasks are hooked, and which are unhooked. Enter:

```
group list all
```

The **pdbx** debugger will display a list similar to the following:

```
0 A    1 U    2 A    3 A
```

Tasks marked with the letter U next to them are unhooked tasks. In this case, task 1 is unhooked. Tasks marked with the letter A are active, hooked tasks. In this case, tasks 0, 2, and 3 are hooked.

RELATED INFORMATION

The **detach** subcommand for the **dbx** command.
The **hook** subcommand for the **pdbx** command.

pmarray

NAME

pmarray – Starts the Program Marker Array, which is an X-Windows tool for monitoring a parallel executable's run.

SYNOPSIS

pmarray

The **pmarray** command starts the Program Marker Array X-Windows tool prior to invoking **poe**. This tool is used for run-time analysis.

FLAGS

None.

DESCRIPTION

The **pmarray** command starts the Program Marker Array. This X-Windows run-time analysis tool consists of a number of small squares, or lights. Each task in a parallel program has its own row of lights. Using calls to the Parallel Utility Functions enables a parallel program to control the appearance of the Program Marker Array Window. Calls to MP_MARKER (for FORTRAN programs) or mpc_marker (for C programs) enables a program to color lights on, and/or send output strings to the Window. Calls to MP_NLIGHTS (for FORTRAN programs) and mpc_nlights (for C programs) enables a program to determine the number of lights displayed per task row.

ENVIRONMENT VARIABLES

This command responds to the following environment variables:

MP_PMLIGHTS Indicates the number of lights displayed per row on the Array. (If, when you invoke **poe**, you override **MP_PMLIGHTS** using the **-pmlights** flag, the Array will redisplay with the new number of lights per row.)

MP_PROCS Indicates the number of program tasks. The **pmarray** command sets the number of task rows displayed in the Array equal to the value of **MP_PROCS**. (If, when you invoke **poe**, you override **MP_PROCS** using the **-procs** flag, the Array will redisplay with the new number of rows.)

MP_USRPORT Indicates the port id used by the Partition Manager to connect to the Array. By default, the Partition Manager connects to the Array using a socket assigned to port 9999. If you get an error message indicating that the port is in use, specify a different port. Standard TCP/IP practice suggests using ports greater than 5000 and less than 10000.

EXAMPLES

To start the Program Marker Array program as a background process, and open its window, enter:

```
pmarray &
```

FILES

/usr/lib/X11/app-defaults/PMarray (Xdefaults file)

/usr/lib/poe/samples/PMarray.ad (X-Windows resource information)

RELATED INFORMATION

Commands: **poe**(1)

Subroutines: **mpc_marker**(3), **MP_MARKER**(3), **mpc_nlights**(3), **MP_NLIGHTS**(3)

poe

NAME

poe – Invokes the Parallel Operation Environment (POE) for loading and executing programs on remote processor nodes.

SYNOPSIS

poe [*program*] [*program_options*]... [**-euidevice** *device_specifier*] [**-euilib** {*ip* | *us*}]
[**-euilibpath** *path_specifier*] [{**-hostfile** | **-hfile**} *host_file_name*] [**-procs** *partition_size*]
[**-resd** {**yes** | **no**}] [**-retry** *retry_interval*] [**-retrycount** *retry_count*] [**-rmpool** *pool_ID*]
[**-savehostfile** *output_file_name*] [**-spname** *name*] [**-cmdfile** *commands_file*]
[**-newjob** {**yes** | **no**}] [**-pgmmodel** {**spmd** | **mpmd**}] [**-labelio** {**yes** | **no**}]
[**-stdinmode** {**all** | *task_ID*}] [**-stdoutmode** {**unordered** | **ordered** | *task_ID*}]
[{**-samplefreq** | **-sfreq**} *samp_int*] [{**-tbuffsize** | **-tbsize**} *buffer_size*]
[{**-tbuffwrap** | **-tbwrap**} {**yes** | **no**}] [**-tmpdir** *temp_trace_directory*]
[{**-tpermsize** | **-tpsize**} *trace_file_size*] [{**-tracedir** | **-tdir**} *final_trace_directory*]
[{**-tracefile** | **-tfile**} *trace_file_name*] [{**-tracelevel** | **-tlevel**} *level_int*]
[{**-ttempsize** | **-ttsize**} *temp_file_size*] [{**-infolevel** | **-ilevel**} *message_level*]
[**-pmdlog** {**yes** | **no**}] [**-euidevelop** {**yes** | **no**}] [**-pmlights** *number_of_lights*]
[**-usrport** *port_ID*] [*fence_string additional_options*]

The **poe** command invokes the Parallel Operating Environment for loading and executing programs on remote processor nodes. The operation of POE is influenced by a number of POE environment variables. The flag options on this command are each used to temporarily override one of these environment variables. User *program_options* can be freely interspersed with the flag options. If no *program* is specified, POE will either prompt you for programs to load, or, if the **MP_CMDFILE** environment variable is set, will load the partition using the specified commands file.

FLAGS

The flags you can specify on this command are used to temporarily override POE environment variables. For more information on valid values, and on what a particular flag sets, refer to the description of its associated environment variable in the ENVIRONMENT VARIABLES section. The following flags are grouped by function.

The following flags are associated with Partition Manager control.

-euidevice	Overrides the **MP_EUIDEVICE** environment variable.
-euilib	Overrides the **MP_EUILIB** environment variable.
-euilibpath	Overrides the **MP_EUILIBPATH** environment variable.
-hostfile or **-hfile**	Overrides the **MP_HOSTFILE** environment variable.
-procs	Overrides the **MP_PROCS** environment variable.
-resd	Overrides the **MP_RESD** environment variable.
-retry	Overrides the **MP_RETRY** environment variable.
-retrycount	Overrides the **MP_RETRYCOUNT** environment variable.
-rmpool	Overrides the **MP_RMPOOL** environment variable.
-savehostfile	Overrides the **MP_SAVEHOSTFILE** environment variable.

-spname Overrides the **SP_NAME** environment variable.

The following flags are associated with Job Specification.

-cmdfile Overrides the **MP_CMDFILE** environment variable.
-newjob Overrides the **MP_NEWJOB** environment variable.
-pgmmodel Overrides the **MP_PGMMODEL** environment variable.

The following flags are associated with I/O Control.

-labelio Overrides the **MP_LABELIO** environment variable.
-stdinmode Overrides the **MP_STDINMODE** environment variable.
-stdoutmode Overrides the **MP_STDOUTMODE** environment variable.

The following flags are associated with VT Trace Collection.

-samplefreq or **-sfreq**
Overrides the **MP_SAMPLEFREQ** environment variable.
-tbuffsize or **-tbsize** Overrides the **MP_TBUFFSIZE** environment variable.
-tbuffwrap or **-tbwrap**
Overrides the **MP_TBUFFWRAP** environment variable.
-tmpdir Overrides the **MP_TMPDIR** environment variable.
-tpermsize or **-tpsize**
Overrides the **MP_TPERMSIZE** environment variable.
-tracedir or **-tdir** Overrides the **MP_TRACEDIR** environment variable.
-tracefile or **-tfile** Overrides the **MP_TRACEFILE** environment variable.
-tracelevel or **-tlevel** Overrides the **MP_TRACELEVEL** environment variable.
-ttempsize or **-ttsize** Overrides the **MP_TTEMPSIZE** environment variable.

The following flags are associated with the generation of diagnostic information.

-infolevel or **-ilevel** Overrides the **MP_INFOLEVEL** environment variable.
-pmdlog Overrides the **MP_PMDLOG** environment variable.

The following are miscellaneous flags:

-euidevelop Overrides the **MP_EUIDEVELOP** environment variable.

-pmlights Determines the number of lights displayed (per row) on the Program Marker Array. This overrides the **MP_PMLIGHTS** environment variable. For more information on the Program Marker Array, refer to the manual page for the **pmarray** command.

-usrport Overrides the **MP_USRPORT** environment variable.

DESCRIPTION

The **poe** command invokes the Parallel Operating Environment for loading and executing programs on remote nodes. You can enter it at your Home Node to:

- load and execute a SPMD program on all nodes of your partition.
- individually load the nodes of your partition with an MPMD job.
- load and execute a series of SPMD and MPMD programs, in individual job steps, on the same partition.
- run non-parallel programs on remote nodes.

The operation of POE is influenced by a number of POE environment variables. The flag options on this command are each used to temporarily override one of these environment variables. User *program_options* can be freely interspersed with the flag options, and *additional_options* not to be parsed by POE can be placed after a *fence_string* defined by the **MP_FENCE** environment variable. If no *program* is specified, POE will either prompt you for programs to load, or, if the **MP_CMDFILE** environment variable is set, will load the partition using the specified commands file.

The environment variables and flags that influence the operation of this command fall into distinct categories of function. They are:

- **Partition Manager control**. The environment variables and flags in this category determine the method of node allocation.
- **Job specification**. The environment variables and flags in this category determine whether or not the Partition Manager should maintain the partition for multiple job steps, whether commands should be read from a file or STDIN, and how the partition should be loaded.
- **I/O control**. The environment variables and flags in this category determine how I/O from the parallel tasks should be handled. These environment variables and flags set the input and output modes, and determine whether or not output is labeled by task id.
- **VT trace collection**. The environment variables and flags in this category determine if and how execution traces are collected for playback using the Visualization Tool (**vt**). They determine which types of traces are collected, and how trace storage is handled.
- **Generation or diagnostic information**. The environment variables and flags in this category enable you to generate diagnostic information that may be required by the IBM Support Center in resolving PE-related problems.
- **Miscellaneous**. The additional environment variables and flags in this category enable additional error checking, and set a dispatch priority class for execution.

ENVIRONMENT VARIABLES

The environment-variable descriptions in this section are grouped by function.

The following environment variables are associated with Partition Manager control.

MP_EUIDEVICE Determines the adapter set to use for message passing. Valid values are *en0* (for Ethernet), *fi0* (for FDDI), *tr0* (for token-ring), and *css0* (for the POWERparallel systems' high-performance switch feature).

MP_EUILIB Determines the CSS library implementation to use for communication – either the IP CSS or the US CSS. In order to use the US CSS, you must have a POWERparallel system configured with its high-performance switch feature. Valid, case-sensitive, values are **ip** (for the IP CSS) or **us** (for the US CSS). The value of this environment variable can be overridden using the **-euilib** flag.

MP_EUILIBPATH Determines the path to the MPL and CSS libraries. This only needs to be set if the libraries are moved. Valid values are any path specifier. The value of this environment variable can be overridden using the **-euilibpath** flag.

MP_HOSTFILE Determines the name of a host list file for node allocation. Valid values are any file specifier. If not set, the default is *host.list* in your current directory. The value of this environment variable can be overridden using the **-hostfile** or **-hfile** flags.

MP_PROCS Determines the number of program tasks. Valid values are any number from 1 to 80. If not set, the default is 1. The value of this environment variable can be overridden using the **-procs** flag.

MP_RESD Determines whether or not the Partition Manager should connect to the POWERparallel system Resource Manager to allocate nodes. Valid values are either **yes** or **no**, and there is no default. The value of this environment variable can be overridden using the **-resd** flag.

MP_RETRY Determines the period of time (in seconds) between processor node allocation retries if there are not enough processor nodes immediately available to run a program. This is only valid if you are using a POWERparallel system Resource Manager. Valid values are any integer greater than or equal to 0. The default is 0 (no retry). The value of this environment variable can be overridden using the **-retry** flag.

MP_RETRYCOUNT The number of times (at the interval set by **MP_RETRY**) that the Partition Manager should attempt to allocate processor nodes. Valid values are any integer greater than or equal to 0. If not set, the default is 0. The value of this environment variable can be overridden using the **-retrycount** flag.

MP_RMPOOL Determines the number of the POWERparallel system pool that should be used by the Resource Manager for non-specific node allocation. This environment variable/command-line flag only applies to the POWERparallel system Resource Manager. Valid values are any identifying pool number, and there is no default. The value of this environment variable can be overridden using the **-rmpool** flag.

MP_SAVEHOSTFILE
The name of an output host list file to be generated by the Partition Manager. Valid values are any relative or full path name. The value of this environment variable can be overridden using the **-savehostfile** flag.

SP_NAME Determines the POWERparallel system Resource Manager to use. This is an identifying name established during POWERparallel system installation. Generally, this is the name of the control workstation.

The following environment variables are associated with Job Specification.

MP_CMDFILE Determines the name of a POE commands file used to load the nodes of your partition. If set, POE will read the commands file rather than STDIN. Valid values are any file specifier. The value of this environment variable can be overridden using the **-cmdfile** flag.

MP_NEWJOB Determines whether or not the Partition Manager maintains your partition for multiple job steps. Valid values are **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-newjob** flag.

MP_PGMMODEL Determines the programming model you are using. Valid values are **spmd** or **mpmd**. If not set, the default is **spmd**. The value of this environment variable can be overridden using the **-pgmmodel** flag.

The following environment variables are associated with I/O Control.

MP_LABELIO Determines whether or not output from the parallel tasks are labeled by task id. Valid values are **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-labelio** flag.

MP_STDINMODE Determines the input mode – how STDIN is managed for the parallel tasks. Valid values are:

all all tasks receive the same input data from STDIN.

a task id STDIN is only sent to the task identified.

If not set, the default is **all**. The value of this environment variable can be overridden using the **-stdinmode** flag.

MP_STDOUTMODE Determines the output mode – how STDOUT is handled by the parallel tasks. Valid values are:

unordered all tasks write output data to STDOUT asynchronously.

ordered output data from each parallel task is written to its own buffer. Later, all buffers are flushed, in task order, to STDOUT.

a task id only the task indicated writes output data to STDOUT.

If not set, the default is **unordered**. The value of this environment variable can be overridden using the **-stdoutmode** flag.

The following environment variables are associated with VT Trace Collection.

MP_SAMPLEFREQ Determines the interval (in milliseconds) at which AIX kernel statistics are sampled when executing a program with tracing on. Valid values are any integer greater than or equal to 0. If not set, the default is 10. The value of this environment variable can be overridden using the **-samplefreq** or **-sfreq** flags.

MP_TBUFFSIZE Determines the size (in bytes) of the buffer used when generating trace files. This may be specified as *nnn*K or *nnn*M. If not set, the default is 5M. The value of this environment variable can be overridden using the **-tbuffsize** or **-tbsize** flags.

MP_TBUFFWRAP Determines that a wraparound storage approach for trace records should be used instead of the default three-tiered approach. With this approach, the system keeps overwriting the buffer instead of flushing it to a temp file. Valid values are either **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-tbuffwrap** or **-tbwrap** flags.

MP_TMPDIR The temporary directory to which output trace files are written. Valid values are any path specifier. If not set, the default is /tmp/*username*. The value of this environment variable can be overridden using the **-tmpdir** flag.

MP_TPERMSIZE Determines the maximum size (in bytes) of the permanent trace file when executing a program with tracing on. This may be specified as *nnn*M or *nnn*G If not set, the default is 10M. The value of this environment variable can be overridden using the **-tpermsize** or **-tpsize** flags.

MP_TRACEDIR Determines the directory to which the final integrated trace file is built. Valid values are any path specifier. If not set, the default is the current directory. The value of this environment variable can be overridden using the **-tracedir** or **-tdir** flags.

MP_TRACEFILE Determines the name of the output trace file created when executing a program with tracing on. Valid values are any file specifier. If not set, the default is the name of the program with the suffix *.trc* added. The value of this environment variable can be overridden using the **-tracefile** and **-tfile** flags.

MP_TRACELEVEL Determines the level of VT tracing that should be generated during the execution of a program. Valid values are:

0 no trace records

1 Application Markers

2 AIX Kernel Statistic and Application Markers

3 Message Passing, Collective Communication, and Application Markers

9 all trace records

If not set, the default is **0** (no trace records). The value of this environment variable can be overridden using the **-tracelevel** or **-tlevel** flags.

MP_TTEMPSIZE Determines the size (in bytes) of the temp file used when generating trace files. This may be specified as *nnn*M or *nnn*G. If not set, the default is 10M. The value of this environment variable can be overridden using the **-ttempsize** or **-ttsize** flags.

The following environment variables are associated with the generation of diagnostic information.

MP_INFOLEVEL Determines the level of message reporting. Valid values are:

- **0** error
- **1** warning and error
- **2** informational, warning, and error
- **3** informational, warning, and error. Also reports diagnostic messages for use by the IBM Support Center.
- **4, 5** Informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center.

If not set, the default is **1** (warning and error). The value of this environment variable can be overridden using the **-infolevel** or **-ilevel** flags.

MP_PMDLOG Determines whether or not diagnostic messages should be logged to a file in */tmp* on each of the remote nodes. Typically, this environment variable/command-line flag is only used under the direction of the IBM Support Center in resolving a PE-related problem. Valid values are **yes** or **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-pmdlog** flag.

MP_PMDSUFFIX Determines a string to be appended to the normal tcp service. The normal tcp service specified in */etc/services* is named *pm2*. By setting **MP_PMDSUFFIX**, you can append a string to *pm2*. If **MP_PMDSUFFIX** is set to *abc*, for example, then the service requested in */etc/services* is *pm2abc*. This permits testing of alternate versions of the Partition Manager daemon. Typically, this environment variable is only used under the direction of the IBM Support Center in resolving a PE-related problem. Valid values are any string. This environment variable has no associated command-line flag.

The following are miscellaneous environment variables:

MP_CSS_INTERRUPT
Determines whether or not interrupts from the high-performance switch are enabled at MPL initialization. This may provide better performance for certain applications. Valid values are **yes** and **no**. If not set, the default is **no**. This environment variable has no associated command-line flag.

MP_EUIDEVELOP Determines whether or not the MPL performs more detailed checking during execution. This additional checking is intended for developing applications, and can significantly slow performance. Valid values are **yes** and **no**. If not set, the default is **no**. The value of this environment variable can be overridden using the **-euidevelop** flag.

MP_FENCE Determines a *fence_string* to be used for separating options you want parsed by POE from those you do not. Valid values are any string, and there is no default. Once set, you can then use the *fence_string* followed by *additional_options* on the **poe** command line. The *additional_options* will not be parsed by POE. This environment variable has no associated command-line flag.

MP_NOARGLIST Determines whether or not POE ignores the argument list. Valid values are **yes** and **no**. If set to **yes**, POE will not attempt to remove POE command-line flags before passing the argument list to the user's program. This environment variable has no associated command-line flag.

MP_PMLIGHTS Indicates the number of lights displayed per row on the Program Marker Array.

MP_PRIORITY Determines a dispatch priority class for execution. See *IBM AIX PE: Installation, Administration, and Diagnosis* for more information on dispatch priority classes. Valid values are any of the dispatch priority classes set up by the system administrator in the file */etc/poe/priority*. This environment variable has no associated command-line flag.

MP_USRPORT Indicates the port id used by the Partition Manager to connect to the Program Marker Array. By default, the Partition Manager connects to the Array using a socket assigned to port 9999. If you get an error message indicating that the port is in use, specify a different port. Standard TCP/IP practice suggests using ports greater than 5000 and less than 10000.

EXAMPLES

1. Assume the **MP_PGMMODEL** environment variable is set to **spmd**, and **MP_PROCS** is set to *6*. To load and execute the SPMD program *sample* on the six remote nodes of your partition, enter:

   ```
   poe sample
   ```

2. Assume you have an MPMD application consisting of two programs – *master* and *slaves*. These programs are designed to run together and communicate via calls to MPL subroutines. The program *master* is designed to run on one processor node. The *slaves* program is designed to run as separate tasks on any number of other nodes. The **MP_PGMMODEL** environment variable is set to **mpmd**, and **MP_PROCS** is set to *6*. To individually load the six remote nodes with your MPMD application, enter:

   ```
   poe
   ```

   Once the partition is established, the **poe** command responds with the prompt:

   ```
   0:host1_name>
   ```

   To load the *master* program as task 0 on host1_name, enter:

   ```
   master
   ```

   The **poe** command responds with a prompt for the next node to load. When you have loaded the last node of your partition, the **poe** command displays the message `Partition loaded...` and begins execution.

3. Assume you want to run three SPMD programs – *setup*, *computation*, and *cleanup* – as job steps on the same partition of nodes. The **MP_PGMMODEL** environment variable is set to *spmd*, and **MP_NEWJOB** is set to **yes**. You enter:

   ```
   poe
   ```

   Once the partition is established, the **poe** command responds with the prompt:

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

   To load the program *setup*, enter:

   ```
   setup
   ```

   The program setup executes on all nodes of your partition. When execution completes, the **poe** command again prompts you for a program name. Enter the program names in turn. To release the partition, enter:

   ```
   quit
   ```

4. To check the process status (using the non-parallel command **ps**) for all remote nodes in your partition, enter:

   ```
   poe ps
   ```

FILES

host.list (Default host list file)

RELATED INFORMATION

Commands: **mpcc**(1), **mpCC**(1), **mpxlf**(1), **pmarray**(1), **vt**(1)

poestat

NAME

poestat – Starts the System Status Array, which is an X-Windows tool for monitoring the operational status and CPU utilization of processor nodes.

SYNOPSIS

poestat [**-norm**]

The **poestat** command starts the System Status Array. This X-Windows tool lets you quickly survey the utilization of processor nodes. Normally, this would be run in the background.

FLAGS

-norm Indicates that the POWERparallel system Resource Manager, which would normally be used to select nodes for monitoring, is not available. Instead, the System Status Array program selects for monitoring each of the nodes on the LAN that have the VT Statistics Collector Daemon (*dig*) running. It also selects each of the nodes listed in the host list file indicated by the **MP_HOSTFILE** environment variable. If you are monitoring nodes of a RISC System/6000 network cluster, you must use this flag.

DESCRIPTION

The **poestat** command starts the System Status Array. This X-Windows analysis tool lets you quickly survey the utilization of processor nodes. The Array consists of a number of squares, each representing a processor node of your POWERparallel system or cluster. The squares are colored pink and yellow to show the instantaneous percent of CPU utilization for each processor node. If a square were to appear all pink, the node would be at 0 percent utilization. If a square were to appear all yellow, it would be at 100 percent utilization. To the right of the Array is a node list which contains the name of each node shown in the Array. The nodes are listed in the order in which they were contacted, left to right, starting with the top row of the Array. You use this list to identify the name of a node represented in the Array.

In order to use this tool, the Visualization Tool's Statistics Collector Daemon process (*dig*) needs to be running on each of the nodes you wish to monitor. The daemon feeds the System Status Array with the CPU information it displays, and is created on each of your nodes as part of the Visualization Tool's installation procedure. The *dig* statistics collector daemon can also feed information to the Visualization Tool. If a square on the Array appears gray, the node is unavailable for monitoring. It either does not have the Statistics Collector daemon running, or the Array cannot communicate with it.

ENVIRONMENT VARIABLES

MP_HOSTFILE This environment variable is normally associated with node allocation. However, it is also checked by the **poestat** command when running with the **-norm** option. It determines the name of a host list file to use in selecting nodes for monitoring. If not set, the default is *host.list* in your current directory.

SP_NAME Identifies the POWERparallel system Resource Manager name, if used.

EXAMPLES

To start the System Status Array as a background process when the POWERparallel system Resource Manager is available, enter:

```
poestat &
```

To start the System Status Array as a background process to monitor a RISC System/6000 network cluster, enter:

```
poestat -norm &
```

FILES

host.list (Default host list file)

RELATED INFORMATION

Commands: **vt**(1)

vt

NAME

vt – Starts the Visualization Tool, which is an X-Windows tool that enables you to visualize performance characteristics of your program and system.

SYNOPSIS

vt [**-tracefile** *trace_file*] [**-tfile** *trace_file*] [**-configfile** *configuration_file*]
[**-cfile** *configuration_file*] [**-spath** *directory_list*] [**-norm**] [**-cmap**] [**-go**]

The **vt** command starts the Visualization Tool for visualizing performance characteristics of a program or the system. This X-Windows tool consists of a group of displays which present specific, often complex, information is easily-interpretable forms such as bar charts and strip graphs. VT can be used to play back traces generated during a program's execution (trace visualization), or as an on-line monitor to study the operational status and activity of processor nodes (performance monitoring).

FLAGS

-tracefile or **-tfile**
: Loads a specified trace file for playback. (A trace file can also be loaded after VT is started.)

-configfile or **-cfile**
: Loads a specified configuration file. A configuration file contains previously saved arrangements of VT windows as well as input field specifications. (A configuration file can also be loaded after VT is started.)

-spath
: Indicates a search path to a program's source code. Like the AIX **PATH** environment variable, this is a series of colon-delimited directory names to search. Unless the program's source is in the current directory, the search path is needed to display it in the VT's Source Code view. (A search path to the program's source code can also be indicated after VT is started.)

-norm
: Indicates that the POWERparallel system Resource Manager is unavailable. In performance monitoring mode, VT normally uses the Resource Manager to learn which nodes are available for monitoring. If this flag is specified, VT instead gets this information from the host list file indicated by the **MP_HOSTFILE** environment variable, and from the LAN. If you are going to use VT for on-line performance monitoring, you must use this flag.

-cmap
: Requests a private color map. If this flag is not used, VT attempts to use the default color map shared by all active X-Windows applications. Depending on the number of active X-Windows applications, there might not be enough available colors for VT. When this happens, VT displays a message indicating the spectrum(s) it cannot allocate, and uses black in place of the unallocated color(s). VT will still run, but in extreme cases some display spectrums may be unusable because of the missing

color(s). When you use this flag, VT makes a private copy of the default X-Windows color map.

DESCRIPTION

The **vt** command starts the Visualization Tool. This is an X-Windows tool for visualizing performance characteristics of your program and system. It consists of a group of displays, or views, which present complex information in easily-interpretable forms such as bar charts and strip graphs. The VT views can be used for trace visualization and on-line performance monitoring.

- In trace visualization mode, VT plays back statistical and event records, or trace records, generated during a program's execution. In this mode, VT can visualize information about the user's program as well as the program's use of the underlying system. The visualized information can help in tuning the program to optimize its use of the underlying system.
- In performance monitoring mode, VT acts as an on-line monitor showing the operational status and activity of each of the processor nodes on a POWERparallel system or RISC System/6000 network cluster. In this mode, VT only displays system statistics and not communication information. In order to use VT for performance monitoring, a Statistics Collector Daemon process named *dig* needs to be running on each of the nodes you wish to monitor. The daemon feeds VT with the AIX kernel statistics it displays, and is created on each of your nodes as part of the Visualization Tool's installation procedure. The *dig* statistics collector daemon can also feed information to the System Status Array started by the **poestat** command.

ENVIRONMENT VARIABLES

MP_HOSTFILE This environment variable is normally associated with node allocation. However, it is also checked by the **vt** command when running with the **-norm** option. It determines the name of a host list file to use to select nodes that are available for monitoring. If not set, the default is *host.list* in your current directory.

EXAMPLES

To load the trace file *mytrace*, the configuration file *myconfig*, and to begin playback immediately upon starting VT:

```
vt -tfile mytrace -cfile myconfig -go
```

To start VT and add the directories */u/files/source* and */u/hink/source* to the source code search path:

```
vt -spath /u/files/source:/u/hink/source
```

To start VT for performance monitoring if you are in an environment where the POWERparallel system Resource Manager is not available:

```
vt -norm
```

FILES

host.list (Default host list file)

/usr/lib/X11/app-defaults/Vt (Xdefaults file)

RELATED INFORMATION

Commands: **poestat**(1)

xpdbx

NAME

xpdbx – Invokes the **xpdbx** debugger, which is the X-Windows interface of the PE debugging facility.

SYNOPSIS

xpdbx [*program*] [*program_options*] [*POE_options*] [X_options] [**-c** [**-f**] [**-l** *directory*]... [**-d** *nesting_depth*] [**-t** *table_size*] [**-x**]

The **xpdbx** command invokes the **xpdbx** debugger, which is the X-Windows interface of the PE debugging facility. This facility is based on the **dbx** debugger, but adds function specific to parallel programming.

FLAGS

The **xpdbx** command accepts standard X-Windows flags. Because the **xpdbx** command runs in the Parallel Operating Environment, it also accepts the flags supported by the **poe** command. See the **poe** manual page for a description of these POE options. Additional **pdbx** flags are:

-f Reads only required symbol table information upon initiation of the **dbx** sessions. You can use this flag to speed startup and to cut down memory usage. This is especially useful when debugging large executable files and when page space is low.

-l (upper-case i)
Adds the specified *directory* to the list of directories searched for an executable's source files. (Once **xpdbx** is running, this list can also be overridden on a group or single node basis using the File Selection Dialog window.)

-d Sets the limit for the nesting of program blocks. The default nesting depth limit is 25.

-t Increases the size of each **dbx**'s type table by the *table_size* number variable. The **-t** flag is needed when an executable file contains more than 5000 types.

-x Prevents the **dbx** command from stripping _ (trailing underscore) characters from symbols originating in FORTRAN source code. This flag enables **dbx** to distinguish between symbols which are identical except for an underscore character, such as xxx and xxx_.

-tmpdir This *POE_option* is normally associated with Visualization Tool trace collection. It specifies the directory to which output trace files are written. For **pdbx**, it specifies the directory to which the individual startup files (.pdbx.*process_id.task_id*) are written for each **dbx** task. This is frequently local, and may be a shared directory. If not used, and if its associated environment variable **MP_TMPDIR** is not set, the default location is */tmp*.

DESCRIPTION

The **xpdbx** command invokes the X-Windows interface of the PE debugging facility. It runs in the Parallel Operating Environment, and is based, and built, on the AIX debugging tool **dbx**

To use **xpdbx** for interactive debugging, you first need to compile the program and set up the execution environment as you would to invoke a parallel program with the **poe** command. Your program should be compiled with the **-g** flag in order to produce an object file with symbol table references. This file is needed for symbolic debugging.

ENVIRONMENT VARIABLES

Because the **xpdbx** command runs in the Parallel Operating Environment, in interacts with the same environment variables associated with the **poe** command. See the **poe** manual page for a description of these environment variables. Additional variables are:

HOME During **xpdbx** initialization, **xpdbx** uses this environment variable to search for two special initialization files. First, **xpdbx** searches for *$HOME/.pdbxinit*, and then *./pdbxinit*.

SHELL The **sh** subcommand in **dbx**, which is available through **xpdbx**, uses this environment variable to determine which shell to use. If this environment variable is not set, the default is the **sh** shell.

MP_DBXPROMPTMOD
The **dbx** prompt *\n(dbx)* is used by **xpdbx** as an indicator denoting that a **dbx** subcommand has completed. This environment variable can be used to modify the prompt. Any value assigned to **MP_DBXPROMPTMOD** will have a "." prepended and then be inserted in the *\n(dbx)* prompt between the "x" and the ")". This environment variable is useful when the string *\n(dbx)* is present in the output of the application being debugged. For example, if **MP_DBXPROMPTMOD** is set to *unique157*, the prompt would be *\n(dbx.unique157)*.

MP_TMPDIR This environment variable is normally associated with Visualization Tool trace collection. In trace collection, it specifies the directory to which output trace files are written. For **xpdbx**, it specifies the directory to which the individual startup files (.pdbxinit.*process_id.task_id*) are written for each **dbx** task. This is frequently local, and may be a shared directory. If not set, and if its associated command-line flag **-tmpdir** is not used, the default location is */tmp*.

EXAMPLES

To start the **xpdbx** debugger, enter:

```
xpdbx
```

The **xpdbx** window automatically opens to mark the start of the debug session.

FILES

.pdbxinit (Initial commands for **xpdbx**)

.pdbxinit.*process_id.task_id* (Initial commands for the individual **dbx** tasks)

/usr/lib/X11/app-defaults/Xpdbx (Xdefaults file)

RELATED INFORMATION

Commands: **dbx**(1), **pdbx**(1), **poe**(1)

"Debugging Programs" in *AIX for RISC System/6000: General Programming Concepts*, (SC23-2205).

Appendix B. POE Environment Variables and Command-Line Flags

This appendix contains tables which summarize the environment variables and command-line flags discussed throughout this book. You can set these variables and flags to influence the execution of parallel programs, and the operation of certain tools. The command-line flags temporarily override their associated environment variable. The tables divide the environment variables and flags by function.

- Table 11 on page 214 summarizes the environment variables and flags for controlling the Partition Manager. These environment variables and flags enable you to specify such things as an input or output host list file, and the method of node allocation. For a complete description of the variables and flags summarized in this table, see Chapter 2, “Executing Parallel Programs” on page 7.
- Table 12 on page 215 summarizes the environment variables and flags for Job Specifications. These environment variables and flags determine whether or not the Partition Manager should maintain the partition for multiple job steps, whether commands should be read from a file or STDIN, and how the partition should be loaded. For a complete description of the variables and flags summarized in this table, see Chapter 2, “Executing Parallel Programs” on page 7.
- Table 13 on page 215 summarizes the environment variables and flags for determining how I/O from the parallel tasks should be handled. These environment variables and flags set the input and output modes, and determine whether or not output is labeled by task id. For a complete description of the variables and flags summarized in this table, see “Managing Standard Input, Output, and Error” on page 34.
- Table 14 on page 216 summarizes the environment variables and flags controlling VT trace collection. These environment variables and flags determine such things as which types of traces are collected, and how trace storage is handled. For a complete description of the variables and flags summarized in this table, see “Generating Trace Files” on page 106.
- Table 15 on page 217 summarizes the environment variables and flags for collecting diagnostic information. These environment variables and flags enable you to generate diagnostic information that may be required by the IBM Support Center in resolving PE-related problems.
- Table 16 on page 217 summarizes some miscellaneous environment variables and flags. These environment variables and flags provide control for the Program Marker Array, enable additional error checking, and let you set a dispatch priority class for execution.

You can use the POE command-line flags on the **poe**, **pdbx**, and **xpdbx** commands. You can also use some of these flags on program names when individually loading nodes from STDIN or a POE commands file. The flags you can use are mainly those having to do with VT trace collection. They are:

- **-infolevel** or **-ilevel**
- **-ttempsize** or **-ttsize**
- **-tmpdir**

© Copyright IBM Corp. 1993

- **-samplefreq** or **-sfreq**
- **-tbuffwrap** or **-tbwrap**
- **-tbuffsize** or **-tbsize**
- **-tpermsize** or **-tpsize**
- **-euidevelop**

In the tables that follow, a check mark (√) denotes those flags you can use when individually loading nodes. For more information on individually loading nodes, refer to "Invoking a MPMD Program" on page 27.

Table 11 (Page 1 of 2). POE Environment Variables/Command-Line Flags for Partition Manager Control

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_EUIDEVICE **-euidevice**	The adapter set to use for message passing – either Ethernet, FDDI, token-ring, or the POWERparallel systems' high-performance switch adapter.	One of the following strings: *en0* Ethernet *fi0* FDDI *tr0* token-ring *css0* high-performance switch	The adapter set used as the external network address.
MP_EUILIB **-euilib**	The CSS library implementation to use for communication – either the IP CSS or the US CSS. In order to use the US CSS, you must have a POWERparallel system configured with its high-performance switch feature.	One of the following strings: *ip* The IP CSS. *us* The US CSS. **Note:** This specification is case-sensitive.	None
MP_EUILIBPATH **-euilibpath**	The path to the MPL and CSS libraries. This only needs to be set if the libraries are moved.	any path specifier	*/usr/lpp/poe/lib*
MP_HOSTFILE **-hostfile** **-hfile**	The name of a host list file for node allocation.	any file specifier or the word NULL.	*host.list* in the current directory.
MP_PROCS **-procs**	The number of program tasks.	Any number from 1 to 80.	1
MP_RESD **-resd**	Whether or not the Partition Manager should connect to the POWERparallel system Resource Manager to allocate nodes. **Note:** This environment variable only specifies whether or not to use the POWERparallel system Resource Manager. The actual POWERparallel system Resource Manager you are using is identified by the environment variable **SP_NAME**. The **SP_NAME** environment variable should have been set as part of the POWERparallel system installation process. If it has not, you need to set **SP_NAME** to the name of the control workstation on the POWERparallel system. For more information on **SP_NAME**, refer to the *IBM 9076 Scalable POWERparallel Systems: SP2 Administration Guide*.	*yes* *no*	None
MP_RETRY **-retry**	The period of time (in seconds) between processor node allocation retries if there are not enough processor nodes immediately available to run a program. This is only valid if you are using the POWERparallel system Resource Manager.	an integer greater than or equal to 0.	0 (no retry)
MP_RETRYCOUNT **-retrycount**	The number of times (at the interval set by **MP_RETRY**) that the Partition Manager should attempt to allocate processor nodes.	an integer greater than or equal to 0.	0

Table 11 (Page 2 of 2). POE Environment Variables/Command-Line Flags for Partition Manager Control

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_RMPOOL **-rmpool**	The number of the POWERparallel system pool that should be used by the Resource Manager for non-specific node allocation. This environment variable/command-line flag only applies to the POWERparallel system Resource Manager.	an identifying pool number.	None
MP_SAVEHOSTFILE **-savehostfile**	The name of an output host list file to be generated by the Partition Manager.	Any relative or full path name.	None

Table 12. POE Environment Variables/Command-Line Flags for Job Specification

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_CMDFILE **-cmdfile**	The name of a POE commands file used to load the nodes of your partition. If set, POE will read the commands file rather than STDIN.	any file specifier.	None
MP_NEWJOB **-newjob**	Whether or not the Partition Manager maintains your partition for multiple job steps.	*yes* *no*	*no*
MP_PGMMODEL **-pgmmodel**	The programming model you are using.	*spmd* *mpmd*	*spmd*

Table 13. POE Environment Variables/Command-Line Flags for I/O Control

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_LABELIO **-labelio**	Whether or not output from the parallel tasks is labeled by task id.	*yes* *no*	*no*
MP_STDINMODE **-stdinmode**	The input mode. This determines how input is managed for the parallel tasks.	*all* all tasks receive the same input data from STDIN. a task id STDIN is only sent to the task identified.	*all*
MP_STDOUTMODE **-stdoutmode**	The output mode. This determines how STDOUT is handled by the parallel tasks.	One of the following: *unordered* all tasks write output data to STDOUT asynchronously. *ordered* output data from each parallel task is written to its own buffer. Later, all buffers are flushed, in task order, to STDOUT. a task id only the task indicated writes output data to STDOUT.	*unordered*

Table 14. POE Environment Variables/Command-Line Flags for VT Trace Collection

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_SAMPLEFREQ **-samplefreq** √ **-sfreq** √	The interval (in milliseconds) at which AIX kernel statistics are sampled when executing a program with tracing on.	an integer greater than or equal to 0.	10
MP_TBUFFSIZE **-tbuffsize** √ **-tbsize** √	The size (in bytes) of the buffer used when generating trace files.	*nnn*K *nnn*M	1M
MP_TBUFFWRAP **-tbuffwrap** √ **-tbwrap** √	A wraparound storage approach for trace records instead of the default three-tiered approach. With this approach, the system keeps overwriting the buffer instead of flushing it to a temp file.	*yes* *no*	no
MP_TMPDIR **-tmpdir** √	The temporary directory to which output trace files are written. **Note:** This environment variable/flag is also used by the **pdbx** and **xpdbx** commands to determine the directory to which individual startup files are written for each **dbx** task.	any path specifier	/tmp/*username*
MP_TPERMSIZE **-tpermsize** √ **-tpsize** √	The maximum size (in bytes) of the permanent trace file when executing a program with tracing on.	*nnn*M *nnn*G	10M
MP_TRACEDIR **-tracedir** **-tdir**	The directory to which the final integrated trace file is built.	any path specifier.	The current directory.
MP_TRACEFILE **-tracefile** **-tfile**	The name of the output trace file created when executing a program with tracing on.	any file specifier.	The name of the program with the suffix *.trc* added.
MP_TRACELEVEL **-tracelevel** **-tlevel**	The level of VT tracing that should be generated during the execution of a program.	one of the following integers 0 no trace records 1 Application Markers 2 AIX Kernel Statistic and Application Markers 3 Message Passing, Collective Communication, and Application Markers 9 all trace records	0
MP_TTEMPSIZE **-ttempsize** √ **-ttsize** √	The size (in bytes) of the temp file used when generating trace files.	*nnn*M *nnn*G	10M
Note: √ symbol denotes flags you can use when individually loading nodes.			

Table 15. POE Environment Variables/Command-Line Flags for Diagnostic Information

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_INFOLEVEL **-infolevel** √ **-ilevel** √	The level of message reporting.	one of the following integers: 0 error 1 warning and error 2 informational, warning, and error 3 informational, warning, and error. Also reports high-level diagnostic messages for use by the IBM Support Center. 4, 5 informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center.	1
MP_PMDLOG **-pmdlog**	Whether or not diagnostic messages should be logged to a file in */tmp* on each of the remote nodes. Typically, this environment variable/command-line flag is only used under the direction of the IBM Support Center in resolving a PE-related problem.	*yes* *no*	*no*
MP_PMDSUFFIX **(no associated command-line flag)**	A string to be appended to the normal tcp service. The normal tcp service specified in */etc/services* is named *pm2*. By setting **MP_PMDSUFFIX**, you can append a string to *pm2*. If **MP_PMDSUFFIX** is set to *abc*, for example, then the service requested in */etc/services* is *pm2abc*. This permits testing of alternate versions of the Partition Manager daemon. Typically, this environment variable is only used under the direction of the IBM Support Center in resolving a PE-related problem.	any string	None
Note: √ symbol denotes flags you can use when individually loading nodes.			

Table 16 (Page 1 of 2). Other POE Environment Variables/Command-Line Flags

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_CSS_INTERRUPT **(no associated command-line flag)**	Whether or not interrupts from the high-performance switch are enabled at MPL initialization. This may provide better performance for certain applications.	*yes* *no*	*no*
MP_DBXPROMPTMOD **(no associated command-line flag)**	A modified **dbx** prompt. The **dbx** prompt *\n(dbx)* is used by the **pdbx** and **xpdbx** commands as an indicator denoting that a **dbx** subcommand has completed. This environment variable modifies that prompt. Any value assigned to it will have a "." prepended and will then be inserted in the *\n(dbx)* prompt between the "x" and the ")". This environment variable is useful when the string *\n(dbx)* is present in the output of the program being debugged.	any string	None
MP_EUIDEVELOP **-euidevelop** √	Whether or not the MPL performs more detailed checking during execution. This additional checking is intended for developing applications, and can significantly slow performance.	*yes* *no*	*no*
MP_FENCE **(no associated command-line flag)**	A "fence" character string for separating arguments you want parsed by POE from those you do not.	Any string.	None

Table 16 (Page 2 of 2). Other POE Environment Variables/Command-Line Flags

The Environment Variable/Command-Line Flag(s):	Set:	Possible Values:	Default:
MP_NOARGLIST **(no associated command-line flag)**	Whether or not POE ignores the argument list. If set to *yes*, POE will not attempt to remove POE command-line flags before passing the argument list to the user's program.	*yes* *no*	*no*
MP_PMLIGHTS **-pmlights**	The number of lights displayed (per row) on the Program Marker Array.	An integer greater than or equal to 0.	0
MP_PRIORITY **(no associated command-line flag)**	A dispatch priority class for execution. See *IBM AIX PE: Installation, Administration, and Diagnosis* for more information on dispatch priority classes.	Any of the dispatch priority classes set up by the system administrator in the file */etc/poe/priority*	None
MP_USRPORT **-usrport**	The port id used by the Partition Manager to connect to the Program Marker Array.	Any positive integer less than 32767. Standard TCP/IP practice suggests using ports greater than 5000 and less than 10000.	9999

Appendix C. Customizing Resources

You can customize certain features of an X-Window. For example, you can customize its colors, fonts, orientation, and so on. This section lists each of the resource variables you can set for the Program Marker Array and the Visualization Tool.

You may customize resources by assigning a value to a resource name in a standard X-Windows format. Several resource files are searched according to the following X-Windows convention:

/usr/lib/X11/$LANG/app-defaults/file_name
/usr/lib/X11/app-defaults/file_name
$XAPPLRESDIR/file_name
$HOME/.Xdefaults

Where *file_name* is *Vt* for the Visualization Tool and *PMarray* for the Program Marker Array. Options in the *.Xdefaults* file take precedence over entries in the preceding files. This allows you to have certain specifications apply to all users in the *app-defaults* file as well as user specific preferences set for each user in their *$HOME/.Xdefaults* file.

You customize a resource by setting a value to a *resource variable* associated with that feature. You store these *resource settings* in a file called *.Xdefaults* in your home directory. You can create this file on a server, and so customize a resource for all users. Individual users may also want to customize resources. The resource settings are essentially your own personal preferences as to how the X-Windows should look.

For example, consider the following resource variables for a hypothetical X-Windows tool:

```
TOOL*MainWindow.foreground:
TOOL*MainWindow.background:
```

In this example, say the resource variable TOOL*MainWindow.foreground controls the color of text on the tool's main window. The resource variable TOOL*MainWindow.background controls the background color of this same window. If you wanted the tool's main window to have red lettering on a white background, you would insert the following lines into the *.Xdefaults* file.

```
TOOL*MainWindow.foreground:    red
TOOL*MainWindow.background:    white
```

© Copyright IBM Corp. 1993

Program Marker Array Resource Variables

The following table details the resource variables you can set for the Program Marker Array.

The resource variable shown here:	Controls:	Possible values:
pmarray*background	The background color of the Program Marker Array window.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
pmarray*WorkArea*background	The background color of the work area on the Program Marker Array window.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
pmarray*fontList	The font used on the Program Marker Array window.	Any of the valid font specifications listed in the file */usr/lpp/X11/lib/X11/fonts/fonts.dir*

VT Resource Variables

The following table details the resource variables you can set for VT.

The resource variable shown here:	Controls:	Possible values:
*.reset.labelPixmap:	The label pixmap for the Return Control Button on the Trace Visualization window.	Name of a bitmap file.
*.stepback.labelPixmap:	The label pixmap for the Step Back Control Button on the Trace Visualization window.	Name of a bitmap file.
*.stop.labelPixmap:	The label pixmap for the Stop Control Button on the Trace Visualization window.	Name of a bitmap file.
*.step.labelPixmap:	The label pixmap for the Step Forward Control Button on the Trace Visualization window.	Name of a bitmap file.
*.play.labelPixmap:	The label pixmap for the Play Control Button on the Trace Visualization window.	Name of a bitmap file.
*.loop.labelPixmap:	The label pixmap for the Loop Control Button on the Trace Visualization window.	Name of a bitmap file.
*.drawnButton1.labelPixmap:	The label pixmap for the Magnification Control on the Trace Visualization window	Name of a bitmap file.
*.PV_View_Selector*background	Background color of the View Selector window.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.runLight.minColor:	The color of the Play Control Button when you are not playing.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.runLight.maxColor:	The color of the Play Control Button when you are playing.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.lag_light.mincolor:	The color of the lag light when off.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*

The resource variable shown here:	Controls:	Possible values:
*.lag_light.maxcolor:	The color of the lag light when lit.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.Zgauge1.troughColor:	The color of the inside of the Trace File Time Control.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.Zgauge1.slider2Color: *.Zgauge1.slider1Color:	The color of the range markers in the Trace File Time Control.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.Zgauge1.pointer:	The color of the triangular pointer below the Trace File Time Control.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.menu1*background:	The background color of the menu bars on the Trace Visualization window.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.event_title.fontlist:	The font used for icon labels on the View Selector window.	Any of the valid font specifications listed in the file */usr/lpp/X11/lib/X11/fonts/fonts.dir*
*.fontList:	The font used on all VT windows.	Any of the valid font specifications listed in the file */usr/lpp/X11/lib/X11/fonts/fonts.dir*
*.delaySlider*pointer:	The color of the Replay Speed Control marker.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.delaySlider*tickGravity:	The placement of the Replay Speed Control tick marks. TRUE places the ticks above the control. FALSE places the ticks below the control.	TRUE or FALSE
*.tracefile:	The default name of the trace file to be played.	Any trace file name. This is overridden by the **MP_TRACEFILE** environment variable and its associated command-line flags **-tracefile** and **-tfile**
*.VT_PMView_Selector.geometry: *.VT_View_Selector.geometry:	The placement of the View Selector window displayed for trace visualization or performance monitoring.	Any standard Motif geometry specification.
*.Dialog*win*geometry:	The placement of any pop-up dialogs.	Any standard Motif geometry specification.
*.NidleColor	The color representing idle time in the CPU load display spectrum. This display spectrum is used by the System Summary view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NuserColor	The color representing user time in the CPU load display spectrum. This display spectrum is used by the System Summary view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NkernelColor	The color representing kernel time in the CPU load display spectrum. This display spectrum is used by the System Summary view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NwaitColor	The color representing wait time in the CPU load display spectrum. This display spectrum is used by the System Summary view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*

The resource variable shown here:	Controls:	Possible values:
*.NnoColor	The color used by the Communication display spectrum to represent the lack of any communication events. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NbsendColor	The color in the Communication display spectrum for events associated with calls to MP_BSEND or mpc_bsend. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NbrecvColor	The color in the Communication display spectrum for events associated with calls to MP_BRECV or mpc_brecv. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NbsendrecvColor	The color in the Communication display spectrum for events associated with calls to MP_BSENDRECV or mpc_bsendrecv. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NsendColor	The color in the Communication display spectrum for events associated with calls to MP_SEND or mpc_send. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NrecvColor	The color in the Communication display spectrum for events associated with calls to MP_RECV or mpc_recv. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NstatusColor	The color in the Communication display spectrum for events associated with calls to MP_STATUS or mpc_status. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NwaitColor	The color in the Communication display spectrum for events associated with calls to MP_WAIT or mpc_wait. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NprobeColor	The color in the Communication display spectrum for events associated with calls to MP_PROBE or mpc_probe. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NenvironColor	The color in the Communication display spectrum for events associated with calls to MP_ENVIRON and mpc_environ. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*

The resource variable shown here:	Controls:	Possible values:
*.NstopallColor	The color in the Communication display spectrum for events associated with calls to MP_STOPALL or mpc_stopall. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NqueryColor	The color in the Communication display spectrum for events associated with calls to MP_QUERY or mpc_query. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NpackColor	The color in the Communication display spectrum for events associated with calls to MP_PACK or mpc_pack. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NunpackColor	The color in the Communication display spectrum for events associated with calls to MP_UNPACK and mpc_unpack. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NbroadcastColor	The color in the Communication display spectrum for events associated with calls to MP_BCAST and mpc_bcast. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NreduceColor	The color in the Communication display spectrum for events associated with calls to MP_REDUCE and mpc_reduce. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NcombineColor	The color in the Communication display spectrum for events associated with calls to MP_COMBINE or mpc_combine. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NscatterColor	The color in the Communication display spectrum for events associated with calls to MP_SCATTER or mpc_scatter. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NgatherColor	The color in the Communication display spectrum for events associated with calls to MP_GATHER or mpc_gather. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NsyncColor	The color in the Communication display spectrum for events associated with calls to MP_SYNC and mpc_sync. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*

The resource variable shown here:	Controls:	Possible values:
*.NshiftColor	The color in the Communication display spectrum for events associated with calls to MP_SHIFT or mpc_shift. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NprefixColor	The color in the Communication display spectrum for events associated with calls to MP_PREFIX and mpc_prefix. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NconcatColor	The color in the Communication display spectrum for events associated with calls to MP_CONCAT or mpc_concat. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NindexColor	The color in the Communication display spectrum for events associated with calls to MP_INDEX or mpc_index. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NpartitionColor	The color in the Communication display spectrum for events associated with calls to MP_PARTITION or mpc_partition. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NgroupColor	The color in the Communication display spectrum for events associated with calls to MP_GROUP or mpc_group. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NgetsizeColor	The color in the Communication display spectrum for events associated with calls to MP_GETSIZE or mpc_getsize. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NgetmembersColor	The color in the Communication display spectrum for events associated with calls to MP_GETMEMBERS or mpc_getmembers. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NgetlabelColor	The color in the Communication display spectrum for events associated with calls to MP_GETLABEL or mpc_getlabel. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*

The resource variable shown here:	Controls:	Possible values:
*.NgetrankColor	The color in the Communication display spectrum for events associated with calls to MP_GETRANK and mpc_getrank. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NgettaskidColor	The color in the Communication display spectrum for events associated with calls to MP_GETTASKID and mpc_gettaskid. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NbvrecvColor	The color in the Communication display spectrum for events associated with calls to MP_BVRECV or mpc_bvrecv. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*
*.NbvsendColor	The color in the Communication display spectrum for events associated with calls to MP_BVSEND or mpc_bvsend. The Communication display spectrum is used by the Interprocessor Communication view.	Any of the valid color specifications listed in the file */usr/lpp/X11/lib/X11/rgb.txt*

Appendix D. Visualization Tool Hints and Tips for Application Developers

This appendix is a separate document included here to supplement the information in Chapter 5, “Visualizing Program and System Performance” on page 93. It is a discussion of the Visualization Tool (VT) and how it works with other components of the AIX Parallel Environment. It also includes a discussion of tools and techniques used in the development process, followed by an introduction on how VT fits into the development process. Some of the frequently used VT displays are also explained.

This knowledge forms a base for understanding how VT is applied. This document assumes that you are familiar with the Parallel Environment, what the VT displays show, and have a basic understanding of the AIX Parallel Environment Message Passing Library (MPL). This document concludes with some important VT operational characteristics that you should know to avoid problems during the development process.

The information presented here are guidelines. Refer to the appropriate chapters in this book for additional description of specific operations. Before continuing, you should do the following:

- Understand the basic concepts of the Message Passing Library
- Execute the VT Quick Operation
- Run VT while reading this document

Introduction

Developing parallel applications is different from developing serial programs. Programming multiple tasks can complicate the effort of making the program operational. Tuning the application requires a thorough understanding of multiple software and system components. The VT provides insights into the programs behavior in a parallel environment.

VT is essentially an animation of the communication and system events that occurred while the program was running. The presentation of data graphically rather that textually allows you to quickly discern differences among parallel tasks. You will be able to easily relate the graphical communication to segments of the application program and detect inefficiencies and abnormalities in the application.

You can use VT throughout the development cycle as an on-line Performance Monitor while the program is executing. You can also use it to:

- Analyze the efficiency of the program after it has executed by replaying trace files.
- Debug the program by isolating faulty sections of code.

VT is useful because it:

- Presents information specifically needed to develop parallel programs
- Helps analyze, tune, and debug parallel application programs
- Helps communicate operational and performance information
- Improves understanding of the programs behavior

© Copyright IBM Corp. 1993

- Helps monitor selected system statistics while an application is running

The components of VT are:

- Tracing Subsystem

  The Tracing Subsystem includes trace collection routines. The Message Passing Library (MPL) calls these routines to record communication events. A statistics gathering program on each node also captures system statistics from the AIX kernel. These records are written to a trace file for later analysis with VT. You can control how many trace records are collected when you start the program or from within the application program.

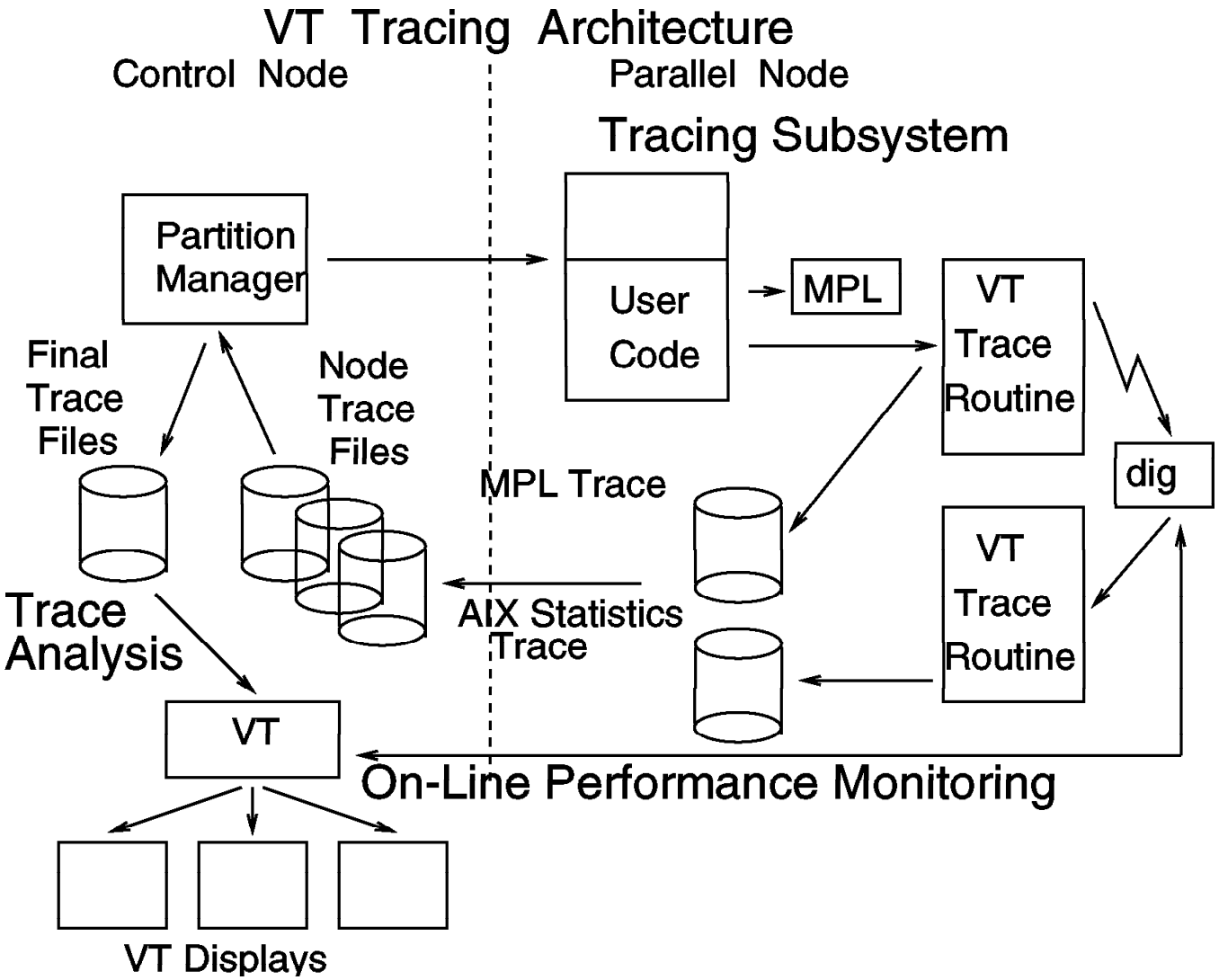
- Trace Analysis (Trace Visualization)

  This is VT's visualization feature that allows you to analyze the program after it has completed running. The information is graphically displayed. You can select a variety of preconfigured displays by pressing a button. The displays are time synchronized so you can understand the sequence of program activity. Communication events and Application Markers are traced for each task. These traces are also collected when a Parallel Operating Environment (POE) job is running multiple tasks per node.

- Performance Monitor

  This is VT's run time visualization feature which provides an On-line Performance Monitor. The Performance Monitor is used while running the application program to obtain a basic understanding of the system resources being used. If tracing is turned on to collect AIX kernel statistics when the program is running, then you can review the same information seen with the On-Line Performance Monitor by using Trace Analysis with VT at a later time. One AIX kernel statistics trace file is collected per node.

The following figure shows the tracing architecture for VT.



From the initiating node, the Partition Manager loads and executes the user code on each node. When a Message Passing Library (MPL) subroutine is called, VT tracing routines are invoked to capture the communication events. These event records are written to a trace file on the node and are eventually written back to the initiating node. When system statistics are being collected, the VT PM node program called *dig* writes this data to the node trace file as well. The *dig* program also delivers the system statistics to VT's On-line Performance Monitor. Each node's trace files are merged into one final trace file after the program completes on all the nodes. Once VT is started, you can select displays of interest. The trace file is read when you press the **play** button on the Trace Visualization Window, thus updating the displays. You can play the VT final trace file from any RISC System/6000 workstation.

Tools and Techniques Used in Developing Parallel Applications

This section describes several tools that help you develop your parallel applications. Each of these tasks lists and discusses the tools that are helpful while developing or modifying an application.

The tools for developing parallel applications are still evolving and there may be some analysis capabilities that VT does not yet provide. Therefore, you may need to augment your development environment with other tools or perhaps write some temporary utilities.

Machine Setup

VT is a stand alone tool and it can be installed on your workstation, on a workstation accessible to a distributed cluster, or on a POWERparallel system. The trace file from a POE run application is transferred to the node where VT resides. After the program completes, you can use the trace file to analyze the program. The cluster or POWERparallel system on which the application program is run, should have its nodes time synchronized. This is necessary for the VT displays to show events among multiple nodes in an understandable and coordinated way. The POWERparallel system's high-performance switch provides a counter that is accessible to all the nodes in the POWERparallel system. VT uses this counter to timestamp events and converts these counter values to the time of day when merging all the node trace files into one. When a POE application program is executed on a machine where VT's tracing programs can find the switch, time synchronization is taken care of for you. However, if a POWERparallel system switch is not present, or if tracing is done in a distributed environment, the system administrator should provide the time synchronization. A public time synchronization tool (NTP) is provided to the customer on a POWERparallel system so that multiple machines can be synchronized. You should install NTP or one of many other available time synchronization tools when a switch is not available to the set of nodes.

On-line Performance Monitoring

VT Performance Monitor

Performance Monitoring provides you with an idea of how the resources are utilized while running an application program. VT's User Load Balance and System Summary displays provide an initial view of processor activity. VT's displays are designed to present data for multiple nodes in a graphical rather than textual format. Information is, therefore, easier to understand and takes up less space. The PM View Selector provides icons for choosing the type of data that you want to display. These views are selected prior to toggling Performance Monitoring on. Other tools, such as IBM's AIX Performance Toolbox/6000 (5696-623 and 5696-624), provide a 3D monitor called **3dmon** that you can also use to display on-line kernel statistics for a parallel system. The following list represents some of the statistics that may interest you during an on-line session.

Constant Statistics:

- Total Real Memory
- Total Virtual Memory
- Total Page Space.

Dynamic Statistics:

- CPU Utilization
- Real Memory available
- Virtual Memory available
- Page Space used or free
- Active processes status
- Disk I/O activity
- Network traffic, such as IP and NFS.

Monitoring Selection By Node or Job

VT's Performance Monitor is intended to be either node or job specific. When you initialize VT and select the Performance Monitor, you need to select which nodes to monitor. When running the application program on a POWERparallel system where the Resource Manager is installed, VT shows jobs that are currently loaded. You select either the job number, job name, or one of the node squares, then select the appropriate VT displays, indicate a frequency at which to collect statistics from the nodes, and then toggle monitoring on. When using Performance Monitoring, you should initially select VT's User Load Balance and System Summary displays. If the POWERparallel system does not have the Resource Manager installed, you need to start VT with the **-norm** command line option. For example:

```
vt -norm
```

This translates to mean that there is no Resource Manager. Therefore, VT does not attempt to communicate with the Resource Manager. Therefore, when you specify this command, job allocation information is not shown on the VT Performance Monitor Window. When selecting a job number, you are effectively selecting the nodes on which that job is running and other work being done on those nodes. The Performance Monitor will broadcast a message on the network and any AIX kernel statistics collection program, such as *dig*, that is running will connect with the Performance Monitor. You can then select these nodes and the nodes listed in the **host.list** file for monitoring.

Resource Monitoring

A parallel application may eventually run out of resources, either caused by the application program or by VT's tracing demands. You should be aware of the amount of resources while running your application. A resource monitor is useful, especially when porting a parallel application because it is not always clear what the demands of the program are. On-line kernel statistics and resource utilization provide a basic understanding and may provide an early warning of a problem that you can rectify instead of having a job abort a few hours after it has begun to run. A resource monitor might identify that one of the parallel nodes temporary disk, such as **/tmp**, is too full prior to starting the application program. Some of the resources to look at are:

- Available temporary directory space, such as **/tmp**

  VT uses a temporary directory to store trace events until it reaches a size limit that you set. You can use the environment variable **MP_TEMPDIR** or the command line option **-tempdir** to override the default temporary directory called **/tmp**. When the temporary trace file limit is met, VT sends the trace file to the directory that you specified from the node where you executed the program. Having a temporary directory that is remote may cause undesired traffic on the network. The temporary directory should exist on the nodes to avoid excessive

network traffic due to tracing. See “Tracing Operation” on page 250 for more information on tracing.

- Available user specified directory space, such as /**home**

  VT collects all the trace information from the nodes to a directory that you specify. If you do not specify one, VT writes it to the current working directory where the program was executed. Prior trace files are saved to a backup directory. In general, tracing requires two times the sum of the trace file sizes collected from the nodes:

  ```
  2 * ((2 * MP_TPERMSIZE) * number of nodes)
  ```

  MP_TPERMSIZE is the limit for two separate trace files collected from each node. Communication and application marker events are collected in one, and AIX kernel statistic trace records are collected in the other. Communication and application marker events are collected per task, while AIX kernel statistics are collected per node. A POE job can, therefore, run several tasks per node but only one AIX kernel statistics trace file is provided per node. See “Tracing Operation” on page 250 for more information on tracing.

System Status Display

The System Status Display is similar to the VT Performance Monitor except that it does not display individual statistics graphically. Rather, it lights up boxes which indicate if there is CPU activity on that node. Refer to the appropriate section in this book for more information.

Application Progress

Applications display or print textual information on the progress and results of the program. These write or print statements in the application program provide information to both the end user and the application developer. The AIX Parallel Environment provides a tool called the Program Marker Array that displays an array of squares that light up with unique colors and print text at the same time by a call to a parallel application subroutine. These application markers are an alternative to using write or print statements. VT traces these markers. VT's Interprocessor Communication and Source Code displays show when the marker events occur and the interprocessor communications display also has a search mode that can help locate the markers when you need them.

With the Program Marker Array, markers use color to show arbitrary program data such as iteration counts and program text buffers. How you decide to use these markers is your decision.

Program Visualization

In addition to the Performance Monitor, VT has a trace visualization mode. This allows you to view the communication activity and the amount of system resources the program is using. The AIX Parallel Environment Message Passing Library subroutines produce communication trace events for VT and for kernel statistics. Both kinds of traces are collected on the nodes and then returned over the network to the directory that you specify. See “Tracing Operation” on page 250 for more information on tracing.

VT's graphical displays help you to better understand the parallel nature of your program by presenting large amounts of data in a meaningful way. The displays record each event in which you are interested. They update themselves at interval

times which you specify to minimize the performance impact of updating the displays. The VT Trace Visualization Window drives the playback action and allows you to closely study the behavior of the program. The Trace Visualization Window **step** button updates the displays for individual events. You can then view and analyze individual or multiple trace files for algorithm characteristics and performance. You might use VT's Interprocessor Communications display first to study the communication characteristics and then migrate to the Source Code display and User Load Balance display as you require more understanding of the programs behavior. "Applying VT" on page 245 covers this topic further.

Debugging

VT can help debug a parallel program by producing a trace file that shows each task's events leading up to the point where an application failed. Some of the situations that might occur while running a parallel program are:

- Abort (such as segmentation violation, disk full)
- Miscommunications (such as receive issued for no corresponding send)
- Communication deadlocks
- Infinitely long execution (such as infinite loop, thrashing)
- A processor stopped functioning

When a program hangs or is taking too long to execute, you can stop the program by typing:

```
control c
```

You can then use VT to show the last sequence of events leading up to the problem.

Some other methods for debugging are:

- Look at printed output
- Program analysis
- Use write or print statements
- Replace MPL subroutines with dummy routines that print the arguments of the message passing library calls
- Use **dbx** on one of the nodes
- Use the parallel debugger called **pdbx**

  The parallel debugger is **dbx** with task management of multiple tasks and uses the standard **dbx** commands.
- Use VT to view the trace
- Use the Program Marker Array and VT to display program markers.

"Debugging Programs During Development" on page 248 provides more discussion on how to use VT and the Parallel Debugger.

Using VT in the Development Process

It is useful to understand how VT and other tools are used while developing parallel applications. Below is a brief introduction to the tasks that are involved and what actions you should consider performing to effectively use the features provided by VT.

Parallel Development with VT

There are many ways to approach developing parallel software. You may be developing a new application or porting an existing one or you may just be interested in understanding the application. The following are several different scenarios that may help you when developing a parallel application:

- Identify the sequence of events leading to the failure of a parallel application.
- Monitor the system activity while the parallel program is executing.
- Identify a communication problem while developing a new parallel application.
- Visualize the parallel task activity to understand an existing parallel application.
- Visualize the application's behavior as the parallelization of an application program scales.
- Compare algorithms for performance and other tuning considerations.
- Understand the asynchronous program behavior in the POWERparallel environment.

Understanding the Parallel Environment

You should understand the environment and how resources and limits are used before executing your application. Executing a program without sufficient resources available may result in a VT trace file being incomplete. Tools should be available to describe the environment's capacities and to show dynamic resources during the execution of a program. You can use the User Load Balance display from VT's Performance Monitor to see how the parallel nodes are used. Some other resources that you can monitor on-line while a program is executing are real memory, virtual memory, paging, CPU, network activity, and temporary disk space. Temporary disk space, such as /**tmp**, on the nodes and the users specified directory or current working directory, such as /**home**, should be monitored periodically. VT uses these when you turn tracing on. Exceeding disk space might prevent VT from recording important trace data. The formulas presented in “Understanding How VT Operates” on page 250 are guidelines to help you estimate disk capacities. You should also understand whether these directories are local or remote. If there are no local directories, writing of the trace records to a remote directory may cause undesired network intrusion. Knowing these values at the time of failure will help you to diagnose and resolve the problems. You can set disk capacities from environment variables or as a command line option when running the application.

Compiling Programs

In order to display the source code with VT, you need to compile the application with the **-g** option which places debug information in the executable. VT uses the executables symbol table, like many symbolic debugging tools, to match traced events with a particular line of code. Currently, the Source code display only supports the Single Program-Multiple Data (SPMD) mode of operating.

If you need to display source code for a portion of a large application program, you can compile select files by using the **-g** flag. However, you also need to compile the C main program or the FORTRAN program with the **-g** flag. When playing the trace file, the Source Code display can hold a particular file for viewing. Normally the file containing the instruction address of the current event being processed by VT is displayed.

Program Setup

Some programs run for a long time. If you turn tracing on when you execute the program, a large trace file may result. You may not want this to happen. Reducing the size of a trace file reduces the amount of time necessary to visualize the trace using VT. "Reducing Trace File Size" on page 249 describes how you can limit tracing to only a particular section of source code. VT supplies these tracing routines as part of the AIX Parallel Operating Environment.

These tracing routines also allow you to select the types of events that you want to trace. You can elect to show communications activity, kernel statistics, application markers (which are similar to print/write statements) independently or all at once. You can also set the sampling frequency for collecting kernel statistics. You might first like to see just the communications activity, and later look at the statistics when you need more of the system information. You can use the Application Markers in the program to monitor progress among the parallel tasks and you can select the types of trace records to collect using environment variables or as a command line option when running the application.

VT uses the following four types of trace records:

- MPL Message Passing

  Each of these events is the result of a call to the MPL Subroutine Library. Trace records are produced when communication constructs such as blocking sends and receives are executed between specific parallel tasks.

- MPL Collective Communication

  Each of these events is the result of a call to the MPL Subroutine Library. Trace records are produced when communication constructs involving all tasks of the parallel program are executed. Examples are broadcasts and combines.

- AIX Kernel Statistics

  These events are extracted from the AIX kernel and include statistics such as:

  - Percent of CPU utilization (user, kernel, wait, and idle)
  - Number of system calls
  - Number of page faults
  - Number of transfers to and from disk
  - Number of blocks read from disk
  - Number of blocks written to disk
  - Number of TCP/IP packets received
  - Number of TCP/IP packets sent.

- Application Markers

  These events result from using either the C subroutine called **mpc_marker** or the FORTRAN function called **MP_MARKER** in the application program. They work in conjunction with VT and the Program Marker Array.

Executing a Program

Executing an application using the Parallel Operating Environment (POE) sets VT tracing off by default. To turn tracing on, use one of the following:

```
-tracelevel
-tlevel
MP_TRACELEVEL
```

Ask yourself the following questions during this phase:

- Did the program run to completion?
- Did the execution produce the correct result?

Depending on your answers, you may need to use VT to isolate the problem and analyze the application. You may also need other tools such as the parallel debugger.

Visualization

Ask yourself the following questions when visualizing:

- Did the program behave as expected?
- What events led to the failure of the program?
- Did the program make efficient use of execution resources?
- Did it behave appropriately when scaling to more tasks/processors?
- How does this execution compare with different algorithms?

The following sections describe how to start VT, what the VT displays show, and how to use VT to understand and analyze performance and isolate an application's communication problems.

Using VT for Trace Visualization

The VT chapter in this book also covers this information. A sample program is also available for new users.

- Compile the program for parallel execution. For C, type:

  ```
  mpcc -g -o program_name program_name.c
  ```

  For FORTRAN, type:

  ```
  mpxlf -g -o program_name program_name.f
  ```

- Run the program with trace options set. For example, the following illustrates restriction of final trace file size and full tracing:

  ```
  export MP_TBUFFSIZE=1M

  export MP_TEMPSIZE=1M

  export MP_TPERMSIZE=3M

  poe program_name -procs 16 -tracelevel 9
  ```

  You can set trace options using command line options or by using environment variables. The options control how often trace information is collected and controls the delivery of trace data to where the program was executed. You can modify the following options:

  – Type of events to collect (MPL Message Passing, MPL Collective Communications, AIX kernel Statistics, and Application Markers)

  - Sampling frequency at which to collect kernel statistics

    such as export MP_SAMPFREQ=500, or every .5 seconds

    The rest of the options can control the amount of network traffic to periodic intervals. See the previous section on how VT operates for additional details.
  - Memory buffer size
  - Temporary trace file size
  - Final trace file size
  - Wrap-around buffer versus sequential write mode for collecting traces
- Program completes

  The trace file, program executable, and source programs are located at the workstation where VT is installed.
- Start VT by typing:

  ```
  vt &
  ```

  You should start VT from the directory where the trace file, program executable, and source programs reside. The PATH environment variable can also be used.
- Select trace file

  Select Tracefile

  Select program_name.trc
- Select Data View(s)

  Use the VT View Selector.
- Play the trace

  Press the **play** button on the Trace Visualization Window.

See the appropriate sections in this book for additional operation instructions.

Using VT for Performance Monitoring

The VT chapter also discusses this information. A sample program is also available for new users.

- Start VT by typing:

  ```
  vt &
  ```
- Select Performance Monitor.

  Select Action

  Select Performance Monitor
- Run the program by typing:

  ```
  poe program_name
  ```
- Select Nodes to Monitor either by specifying a job, a node name, or a node square on the Performance Monitoring Window.
- Select Data View(s) using the PM View Selector

- Specify the Sampling Frequency

  This value is set at an interval that prevents excessive updates to your X Terminal. Do not lower the interval if you are monitoring many nodes or excessive network traffic and display refreshing will occur and the workstation may operate slowly.

- Start Monitoring

  Select File

  Select Toggle Monitoring

  The displays should now begin to show data.

See the appropriate sections in this book for additional operation instructions.

Contents of the VT Graphical Displays

You will need to select one or more VT displays soon after invoking VT. You can choose the Collective Communication, User Load Balance, and Source Code displays to start. As you learn how to use VT and the data it presents, you can collect more information while the program executes and, therefore, gain more understanding of the program's behavior.

VT offers many displays that show particular aspects of a programs behavior or the behavior of associated system resources. These displays are described here to provide a better understanding of their differences. The VT chapter in this book describes the displays in more detail.

General Types of Displays

The two types of VT displays are:

- Streaming Displays

  Examples of these displays are the Processor Utilization (stripgraph type) and Interprocessor Communication displays. These display a historical record of events as time progresses. The time represented under the "current time line" shown in these type of displays corresponds to the current time on the Trace Visualization Window.

- Instantaneous Displays

  Examples of these displays are the Processor Utilization and (bar type) User Load Balance. These display current event values that correspond to the current time on the Trace Visualization Window.

Communication/Program Displays

Each of the Communication displays (Interprocessor Communication, Connectivity Graph, and Message Status Matrix) show both MPL Message Passing and MPL Collective Communication messages. MPL Message Passing is interpreted as the paths the messages take. MPL Collective Communication, however, displays how the information flows. The following discussion covers the MPL Message Passing representation and the MPL Collective Communication representation.

- Interprocessor Communication (MPL Message Passing)

  The Interprocessor Communication display shows individual communication states on each task, as well as abstract communication events. An abstract

communication event lasts from the point at which an application task knows it is sending data (the beginning of a send) to the point where another application task knows the data has been received (the end of the blocking receive, the wait (mpc_wait), or the status (mpc_status) associated with a receive. The abstract communication event is shown by a line drawn over its duration from the sending task to the receiving task but it is not drawn until it is known to be completed.

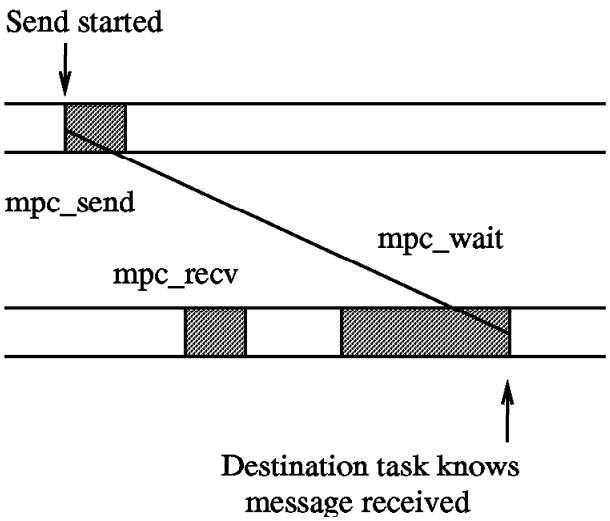
Interpreting the communication lines requires knowledge of the AIX Parallel Environment Message Passing Library.

These are the scenarios for drawing the communication lines:

- The beginning of a blocking send is drawn to the end of a blocking receive. The line is drawn as soon as both have finished.
- The beginning of a non-blocking send is drawn to the end of the successful wait or status MPL call. The line is drawn when both the send, and the wait or status have finished.

  The wait completes when the receipt of the message has actually happened.

  A non-zero value indicates a successful status showing the length of the message. A zero return value indicates that the message is pending.
- The beginning of a send (blocking or non-blocking) is drawn to the end of the successful MPL call that verifies its receipt. The verification can be by blocking receive, wait or status. The lines are drawn as indicated above.



You can also analyze communications further by searching for events with the Interprocessor Communication display search feature. This is selected using the mouse 3 button (right mouse button). This feature allows task numbers and communication event types to be targeted. When the searched event is found, that event is placed under the time line. You can also press the **skip** button to skip through a series of events on selected tasks.

- Connectivity Graph (MPL Message Passing)

  The Connectivity Graph display, like the Message Status Matrix display, shows pairs of tasks involved in communications. This differs from the Interprocessor Communications display slightly. The Interprocessor Communications display

connects lines between the beginning of the send to the end of the MPL call where the destination task knows the message is received. The Connectivity Graph on the other hand, draws arcs between tasks representing the Message Passing messages (straight lines are used to represent a Collective Communication group). The arcs are drawn between tasks during the time the send and receives are posted. For example, for a non-blocking send (mpc_send), the arc between tasks is drawn at the start of the mpc_send and erased at the end of the mpc_send. The same is true for mpc_recv. The arc is not drawn during the wait or query. This allows you to determine the time period between when the receive was posted and when the application program began to use the data in the message.

This display represents both Message Passing and Collective Communication subroutine calls. Message Passing calls are drawn as arcs and the Collective Communication calls appear in straight lines. This display also has a cumulative mode that keeps the lines present even after being received so the network topology is shown. The task circles can be moved to show a more logical representation.

- Message Status Matrix (MPL Message Passing)

  The Message Status Matrix display, like the Connectivity Graph display, shows pairs of tasks involved in communications. This display works in the same manor as the Connectivity Graph display. The Message Status Matrix, however, colors squares of the source/target array whenever a send or receive operation is being handled by the Message Passing Library. Only one square will light representing the send/receive pair. This differs from MPL Collective Communications where all the tasks within the group will be lit. The sources are the rows and the targets are the columns.

Relationships Among Communication Displays

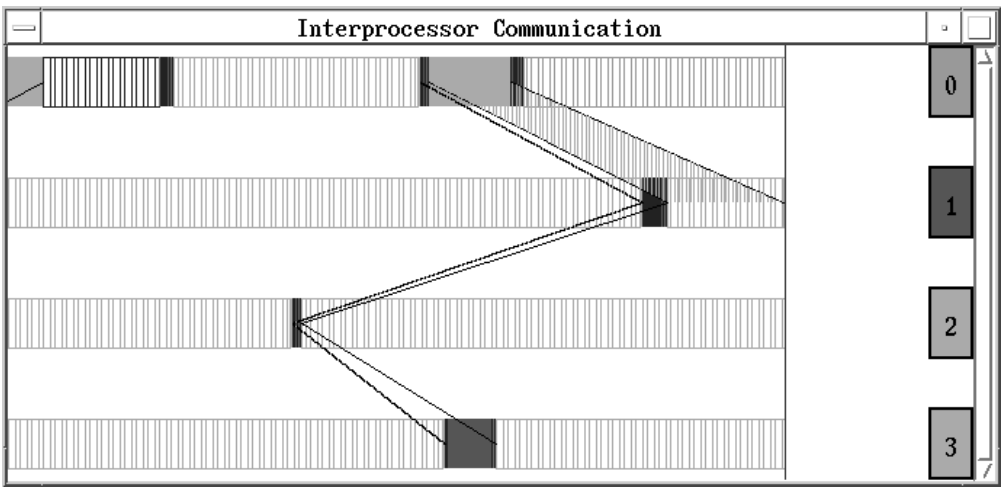
The Message Status Matrix and the Connectivity Graph are useful when the Interprocessor Communication display has drawn the send line but not the line to the receive destination task. The Message Status Matrix and the Connectivity displays indicate who the intended receivers were. Both displays depict Message Passing differently from Collective Communications. Many sends closely packed on the Interprocessor Communications display could be differentiated using the other two communication displays (such as identifying intended receivers). When several squares of the Message Status Matrix are displayed, the Connectivity Graph would indicate if the tasks were involved in Collective Communications (group) or Message Passing. The Message Status Matrix identifies the source task for a receive and the destination task for a send, whereas the Connectivity Graph does not show this. You can use the Interprocessor Communications display to compare system activity when an MPL call is being shown.

MPL Collective Communications

MPL Collective Communications are displayed differently than MPL Message Passing communications. MPL Collective Communications are drawn if the members of the task group are sharing information. All members of the group are using the same instruction. This is unlike the MPL Message Passing communication calls where there can be several different ways to know when the receive has completed.

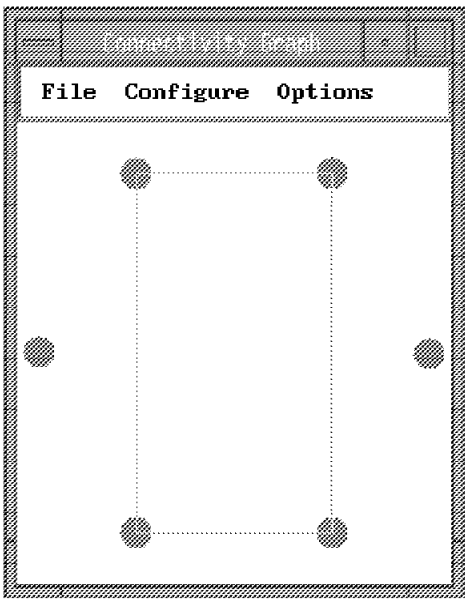
- Interprocessor Communication (MPL Collective Communication)

This display constructs a polygon among all the tasks in the group. A line is drawn from the beginning of the event on the first task to the beginning of the event on the next task and so on until all the tasks in the group are connected. A similar line is drawn connecting the end of all the tasks, thus forming a polygon among all the tasks in the group. The polygon connecting all the tasks is then shaded. Intersecting communication lines are visible through the shaded region.



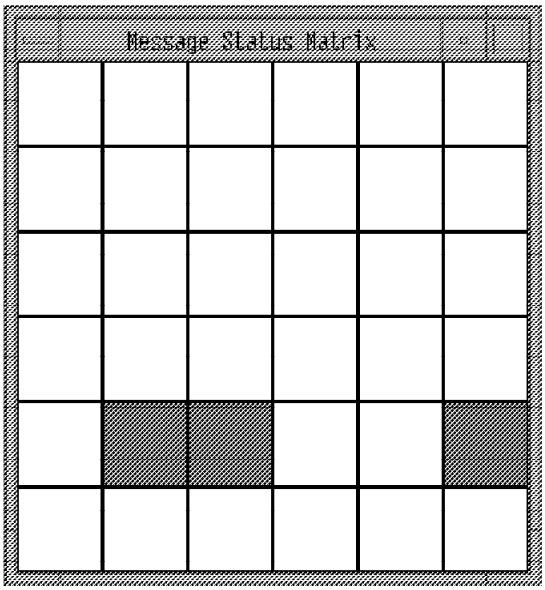
- Connectivity Graph (MPL Collective Communication)

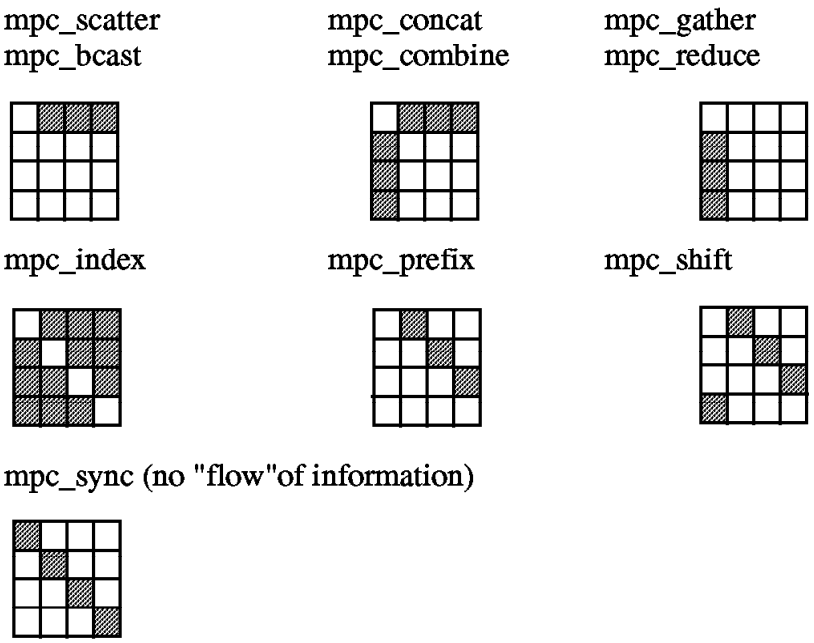
  Collective Communications on this display connects the tasks in the group with a straight line, forming a polygon. The straight lines are used for Collective Communications, whereas arcs are used for Message Passing communications. The arcs connect the source/destination pair and the lines show the group of tasks sharing information. The straight lines among the tasks are displayed as long as one of the MPL Collective Communication instructions is active.



- Message Status Matrix (MPL Collective Communication)

The Message Status Matrix colors squares of the source/target array whenever any task within the group is being handled by the Message Passing Library. All the tasks involved within the group are lit except the source. A broadcast would light up the target tasks belonging to the group during the time the Interprocessor Communications display is drawing an **mpc_bcast** event on any of the tasks. The **mpc_sync** is drawn by coloring the diagonal squares of the array for the tasks involved in the group. This indicates that there is no flow of information. The *IBM AIX Parallel Environment: Programming Primer* provides illustrations that you can use to relate the Message Status Matrix display with each of the MPL Collective Communication groupings. Some examples are shown below:





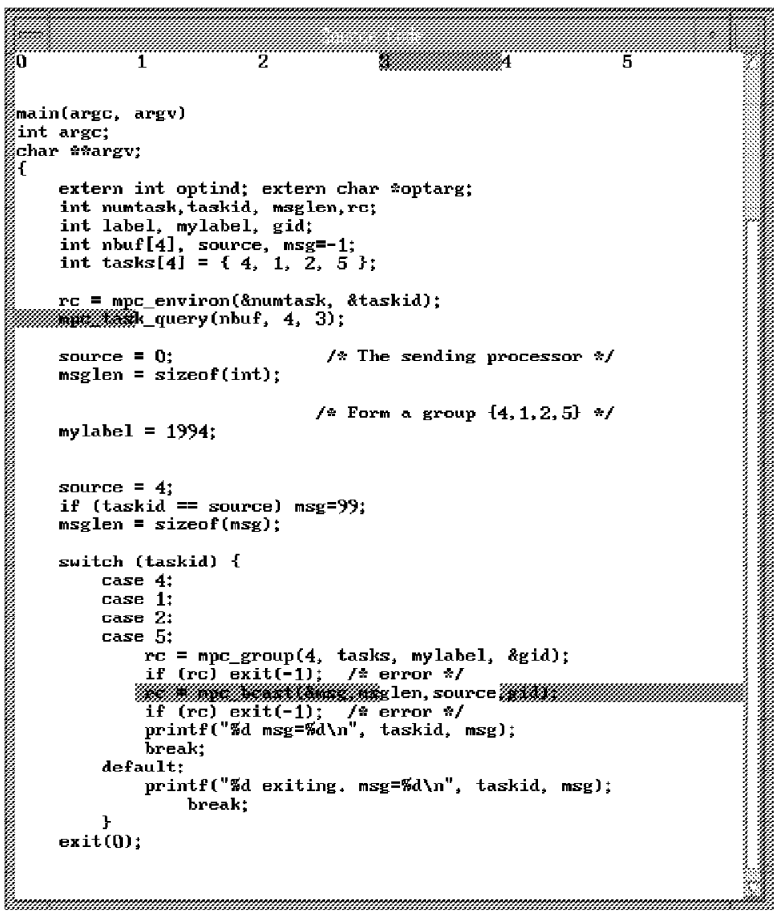
Example of CCL events involving all tasks (eg. allgrp)

The remaining display shows the program source code. All communication events and application markers are highlighted in this display.

- Source Code

  This display identifies the last communication event processed by VT for each task. Each line of the source display is segmented in *n* segments, where *n* is the number of tasks. When VT processes a communication event, it lights up the appropriate segment. This display is often correlated with the Interprocessor Communication display. You can press the **step** button to determine the program's location and communications sequence over a selected range of events. This helps you to determine which section of code is being executed by knowing which communication calls are currently being processed or which ones were just started.

  Additional Source Code display options, like the cumulative mode, are discussed in “Debugging Programs During Development” on page 248.



Processor Utilization Displays

- System Summary

  This display shows the percentage of CPU activity for each node, including user, kernel, wait, and idle activity. This display is useful for determining the amount of CPU time spent or devoted to working on the application compared to the amount of time spent waiting for I/O to complete.

- User Load Balance (one spoke per node)

  This display shows the applications contributions to CPU activity across the system. Each spoke of the wheel represents the amount of CPU utilized (0-100%) for each node. This value represents the user statistic only. The System Utilization display and the individual utilization graphs show the user, kernel, idle, and wait statistics. The user statistic represents the applications computational activity plus the work done by the message passing library for communications. The average utilization is also displayed and is displayed as a shaded polygon. The average utilization is calculated from the first record read by VT but is not displayed until a meaningful average is calculated. When using the VT Performance Monitor, toggling monitoring off and on will reset the average calculation. Comparing the instantaneous values and the average load balance values can help one to balance their parallel application.

There are several other displays that you can select from VT's View Selector. The ones mentioned previously are covered in particular because how they appear and operate may be new to you. This book contains a description of all of the displays.

Refer to the appropriate sections for details. A statistical analysis feature is provided on all graph displays. These statistics help isolate differences among individual tasks and differences from an average across all tasks.

Applying VT

This section covers particular situations where you can use VT to create, modify, port, or understand a parallel program.

Visual Understanding of the Program's Operation

This section covers some display characteristics that help evaluate and understand the information contained in the VT trace file.

VT can help you to understand and tune an application. Often times, however, getting the application to run can require you to spend time identifying the cause of errors in your parallel application. “Debugging Programs During Development” on page 248 covers these situations. When you first begin to use VT, run your application with VT tracing on while also viewing the on-line activity with VT's Performance Monitor. You can then use VT to replay the trace file to understand the program's behavior.

There are several displays available with VT. Refer to “Contents of the VT Graphical Displays” on page 238 for a description of each display.

You should pay close attention to the following characteristics:

- Communication Patterns and Characteristics

  The Interprocessor Communications display shows lines depicting messages between tasks. Patterns quickly emerge for different algorithms. These patterns, combined with the study of the source code, help you to identify which section of code is being executed. Correlating other VT displays with Interprocessor Communications display is an excellent starting point for understanding the application program.

- Processor Load Balance

  VT's User Load Balance display shows both the instantaneous and average user CPU utilization for each node. A program would typically want to make equal use of the available tasks, forming a round pattern on the User Load Balance display. Generally, the more "rounded" the average values look the more balanced is the application. You should also examine the size of the average values. This will indicate how effective the algorithms and program structure are using the CPU resources. An effective use of CPU resources is indicated by larger average values. You can compare lull periods of instantaneous activity with the Interprocessor Communication display to tune the application.

- Computation versus Communication

  VT's User Load Balance, System Summary and Processor Utilization displays provide a good overview of processor utilization across all nodes. The "user" statistics represent both the computation by the user program and the processor overhead for communications. With VT's Interprocessor Communications display, you can correlate the instantaneous CPU utilizations with the communication activity at the "current time line". This helps to

differentiate program computation from communication overhead. You can also view the Source Code display to help identify computational sections of code.

There may be moments where additional communications are necessary. For example, the program may have some conditional logic that causes special processing of messages to occur. You may not want this to happen but it may be necessary for the algorithm. In this case, you may have to closely study communications versus computation.

- Timing characteristics

  You can obtain detailed information about the events, including timing characteristics, by pressing the Mouse 1 button (left mouse button). You can also examine the trace file records. A description of the records appears in an appendix of this book. Analyzing the trace file records could provide summary or detailed characteristics of the specific application execution.

Analysis and Tuning Considerations

The main advantage to a parallel program versus a serial program is that it processes faster. Achieving an efficient program requires that you learn how to use the computational and communication resources in the most effective way. This section describes how VT can help you to identify the efficiency of your program. “Debugging Programs During Development” on page 248 covers the different problems that may arise and how to use VT to isolate the problem.

You can use the following displays to evaluate computational effectiveness:

- Processor Utilization
- User Load Balance
- System Summary.

These displays help to identify idle processors. There are four Communication / Program displays which help you to evaluate the behavior of your code. They are the:

- Interprocessor Communications
- Message Status Matrix
- Connectivity Graph
- Source Code displays.

There are additional displays that show resource activity about the system, network, and disks. These become important when these resources impact communications or processor utilization.

You should try to identify where work is being held up because it is waiting for something else to occur.

Some VT display characteristics to look for are:

- Processor idle time

  Any of the processor utilization displays can help identify low periods of CPU usage. If the CPU utilization is constrained, it may be due to high network load.

- Inappropriate use of blocking versus non-blocking calls

  When processor usage is lower than expected, it may be due to one or more tasks waiting for messages rather than processing. Use non-blocking sends

and receives to reduce waiting and improve performance. Instead of using a wait, you may want to use a loop that does useful work and checks the status at each iteration. You may also want to use status at the end of a loop rather than issuing waits.

- Lengthy messaging operations

  Each Message Passing or Collective Communication Library call produces two trace records, one at its start and another upon return to your program. A bar on the Interprocessor Communications display represents the time spent in the library routine. This is the time when the processor is unavailable to the application even though it may have other work it could be doing. Long bars can indicate points where processor time is expended without advancing the application. Consider changing your algorithm if it will shorten these bars.

  Communication operations fall into three classes:

  – The non-blocking send, receive and status library calls return the processor to the application immediately and should not produce long bars. These operations post or query a communication but do not retain the processor until the communication completes.

  – The blocking send, receive, and wait library calls do retain the processor until the operation completes and can result in expensive delays. If long bars occur at points in the application where the processor could be doing useful work which does not depend on the completion of the operation in question, you might want to try and improve the algorithm. Non-blocking messaging followed by independent work and occasional status checks may produce better resource utilization.

    If you are using blocking operations and you often see long bars, there are two plausible causes for long delays; the communication system may be overloaded or the single task at the other end of the operation may not have completed yet.

  – The collective communication library calls also retain the processor until the operation completes. View these operations as potentially synchronizing across all involved tasks. If the participating tasks are not more or less in step, then all but one task may do some waiting and you will see long bars. If it is practical to do the collective communication when the tasks are more likely to be in step, it may be worthwhile. Collective communication delays can also result from an overload on the communication subsystem.

- Varying user load

  The User Load Balance display instantaneously indicates how much CPU each node is using. Keep track if the node usage fluctuates excessively or one node's usage is very different from another node's usage.

- Long periods without communication

  All of the communication displays show when no communication is taking place. If CPU usage is low during these periods, then you may be able to improve your code. The source code display can suggest where you are in a program.

- Lack of communication on some tasks

  The communication displays may indicate a processor is not being used. This may be a deficiency in your algorithm or an error. The Connectivity Graph also has a cumulative mode that allows the tasks to be graphically configured to

represent a particular network topology. This could help show incorrect send/receive tasks (such as the messages are not going to the correct task).

- Excessive communications

  The Message Status Matrix illustrates how much application time is devoted to messaging. If there is too much time devoted, this may indicate ineffective message passing. Also check other indicators to see if the system resources are being used effectively and if communications are being managed properly.

- Difference in execution time of multiple algorithms

  You can invoke VT multiple times to compare characteristics of different algorithms. The Interprocessor Communications display can be helpful in determining the relative time differences between the two algorithms.

There are many other program characteristics that you can learn about from the VT displays. As you become more familiar with VT and the behavior of your program, identifying these and other characteristics will become faster and easier.

Debugging Programs During Development

Before optimizing an application program, it must first be able to run. When developing a parallel application, there may be many problems that you will need to resolve to successfully execute the program. Several types of problems and general debugging techniques are presented in “Debugging” on page 233.

To use VT to help debug, you should know how to create a trace file when a problem occurs. If you turn tracing on when you execute the program and an error occurs during processing, you can use the trace with VT to isolate the problem. In some situations, you may need to issue **Control-c** to terminate the program. You can use VT to show the events leading up to the point where the problem occurs. Use of the displays is covered in “Analysis and Tuning Considerations” on page 246 and “Contents of the VT Graphical Displays” on page 238. You select the appropriate displays and play the trace file to the end. A quick way to go to the end of the trace file is to drag the Trace File Time Control on VT's main control panel to the 100 percent mark. This will show the latest events leading up to the problem.

Problems like blocking receives or waits for which no send has occurred will be quite apparent. Blocking receives and waits can hang a program but non-blocking receives will not hang a program. There are other problems that will take a little more thought and investigation. An example is when a send from one processor might never get to its destination. You can use VT to identify the message passing library call or section of code that you believe is causing the problem by using the Interprocessor Communication and Source Code displays, but you may need to use the AIX Parallel Environment Parallel Debugger (**pdbx**) to display the value of dynamic variables at the time of execution.

VT's Source Code display has a unique feature that can help identify progress through the program. The Source Code display has a cumulative mode. When selected, it will identify the number of times that your program executes a particular message passing library call. The cumulative mode can even tell you if a particular call has been executed at all. In cumulative mode, once the call has been executed, it will stay lit for the processor that executed it.

Reminder

To see your source code using VT, you must compile the application with the debug flag set (-g).

"Reducing Trace File Size" covers some additional techniques to isolate problems by limiting the scope of what gets traced.

Summary statistics can also help provide some understanding of the application and may provide some use in debugging situations. VT does not offer these but it is possible to examine the trace file records. A description of the records appears in an appendix of this book. Analyzing the trace file records can provide statistics such as average communication time for specific message passing library calls, listing of message sizes (average, minimum, maximum), listing of the events prior to a problem in the program, level of tracing (such as communications only, communications and statistics), and much more.

Reducing Trace File Size

Some applications are very big and may produce trace files that are too large. The following lists describes some ways for reducing the size of a trace file:

- Use selected trace levels

  Tracing routines are provided by VT and are listed with the AIX Message Passing Library calls. See the *AIX Parallel Environment: Parallel Programming Subroutine Reference* for the syntax of these routines. The level of tracing should be identified first. You can do this at program execution time or from within the application. At execution, the **MP_TRACELEVEL** environment variable will be overridden by a **VT_trc_set_params_c()** call within the code. These controls select message passing, collective communications, kernel statistics, markers or a combination of these events to be traced. For example, you could start out with communication trace records only and then add kernel statistics. You can also begin by requesting a sample of the kernel statistics at a wide frequency and later narrow your request to obtain more information.

- Adjusting frequency of the kernel statistics traced

  See the previous discussion. The environment variable **MP_SAMPFREQ** is overridden by the values set in **VT_trc_set_params_c()**. Widening the sampling frequency also reduces the trace file size. Details on statistics is traded for trace file size.

- Isolate when tracing turned on

  You should initially turn tracing off since it is on at the start of the program. At the appropriate points in the code, you can start and stop tracing. You should turn tracing on and off only at task synchronization points within the code. Failure to do so may result in traces for one processor but not another. You can use the **mp_sync()** Message Passing Library call if needed. Also, you should not turn tracing on in a tight loop of code. This would create many trace records and extend the time it takes to display the trace file with VT.

Using VT to display sparse trace files requires you to understand how VT operates. VT shows events as if they were played in real time. A one second run takes VT one second to display. Therefore, during the untraced segments of time, you should stop the trace file. After stopping the trace file, press the **step** button to

search for the next event. Then, press the **play** button to continue normal processing of the trace file.

Understanding How VT Operates

Tracing Operation

An overview of VT tracing is provided in the VT chapter of this book. The following information describes what you should know before tracing with VT.

VT's traces are first written to a buffer on each node and then written to a temporary directory, such as /**temp**. The temporary trace records eventually are written to a directory where the program was started. Each of these storage areas are limited in size. Each has an associated command line argument and environment variable that you can set. The following summarizes these storage locations and their environment variable:

- Temporary Buffer Size - **MP_TBUFFSIZE**
- Temporary trace file size - **MP_TTEMPSIZE**
- Permanent trace file size - **MP_TPERMSIZE**

You should set:

- **MP_TBUFFSIZE** less than **MP_TTEMPSIZE**
- **MP_TTEMPSIZE** less than **MP_TPERMSIZE**

Note: The temporary trace file directory is intended to be a local disk, such as /**tmp**.

There are two temporary trace files created; one file holds AIX kernel statistics records and the other holds communication and application marker event records. Each are limited by **MP_TTEMPSIZE**. When the temporary trace file reaches its limit, the records are written to the permanent trace file in the specified user directory. The temporary trace files for each task (node) are catenated to the permanent trace file until the **MP_TPERMSIZE** is reached, representing each of the two trace file types (communication events and AIX statistics). Once the permanent trace file reaches its limit, all subsequent trace records are purged. Therefore, you should set a limit appropriate for the program being traced. The permanent trace files from each node get merged at the completion of the program.

Note: You should size the temporary trace file to reduce the frequency of network traffic and avoid intrusion to the executing program.

You should also know how trace files are merged. A permanent trace file for each execution node is written to the directory that you specify. Four nodes means eight trace files end up in the directory that you specify. This means that during merge processing, a four node run requires the space for about 16 permanent trace files. The permanent trace files are essentially duplicated when merging takes place. Therefore, the general rule is to have the following space available in the user specified directory:

- User specified directory space needs to be greater than or equal to 2 times the product of permanent trace files times the number of nodes:

  ```
  2 * ((2 * MP_TPERMSIZE) * number of nodes) )
  ```

remembering that there are two types of permanent trace files produced for each node. If the POE job uses multiple tasks per node, there will be fewer AIX kernel statistic trace files produced than communication event trace files. One AIX kernel statistic trace file is created per node.

Therefore, an 8 way job that has **MP_TPERMSIZE** set to 5 Megabytes could produce a trace file of about 80 Megabytes but needs 160 Megabytes to account for merge processing.

You should ensure that:

- The permissions of the temporary directory (such as /**tmp**) and the users specified directory (such as **usd**) where the nodes permanent trace files get placed is set so that the application program can write to these directories.

The current working directory is the user specified directory by default. You can redirect where the permanent and final trace files are written by using **MP_TRACEDIR**.

VT checks that there is sufficient space on the temporary disk prior to writing to the temporary trace file. If the directory is full, VT flushes the temporary trace file to the permanent trace file and cuts **MP_TTEMPSIZE** in half. This, however, does not prevent the application from writing to a nearly full disk. For instance, if the disk is almost full, the application could write to the disk in the meantime, thus causing a disk full error. You should understand both the tracing requirements and the applications demand on resources.

Instructions for using VT in an AFS** environment are described in the **/usr/lpp/vt/README.afs** file.

VT Display History

All of VT's displays remember a select number of events, called the *history buffer*. The history buffer in a streaming display is normally larger than the non-magnified display window. This buffer is eventually filled during normal play. Once the buffer is full, new events read from the trace file replace previously displayed events. When stepping backwards through the trace or moving the Trace File Time Control line on VT's main control panel or demagnifying time, displays like Interprocessor Communications may be clipped at some earlier time. This is where the history buffer ends. To view events earlier in time, you need to replay the trace from the beginning. Therefore, there is a limited range of time for viewing recent events. An alternative is to limit collection of traces to defined sections of the program. You can accomplish this by using trace control subroutines which are covered in “Reducing Trace File Size” on page 249.

Glossary of Terms and Abbreviations

This glossary includes terms and definitions from:

- The *Dictionary of Computing*, SC20-1699.
- 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) and *IBM DATABASE 2 Application Programming Guide for TSO Users* (SC26-4081).

© Copyright IBM Corp. 1993

A

address. 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. It is a graphical user interface (GUI) for the RISC System/6000. It has the following components:

- A graphical user interface and toolkit based on OFS/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.

application. The use to which a data processing system is put; for example, a payroll application, an airline reservation application.

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.

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 provides excellent price/performance.

D

default. An alternative value, attribute, or option that is assumed when none has been specified.

F

file. * A set of related records treated as a unit, for example, in stock control, a file could consist of a set of invoices.

foreign host. Any host on the network other than the local host.

G

GUI. Graphics User Interface, providing multiple windows and iconic representations of system capabilities. Users typically select system services using a mouse to guide a sprite to point and click on a selected icon.

H

host. A computer connected to a network, and providing an access method to that network. A host provides end-user services.

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" and is protected from user tampering by the hardware.

M

menu. A display of a list of available functions for selection by the user.

message passing. The method of communication among processor nodes operating in parallel with distributed memory.

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

Motif. Usually called OSF/Motif. The UNIX industry's standard user interface, originally developed by the Open Software Foundation (OSF). Motif is based on the X-Window system and is a Presentation Manager look-alike. Motif is available for all IBM AIX workstations.

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. In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network.

nodeid. The specific symbolic name assigned to a node during network definition.

P

parameter. * (1) A variable that is given a constant value for a specified application and that may denote the 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.

process. * (1) A unique, finite course of events defined by its purpose or by its effect, achieved under defined conditions. * (2) Any operation or combination of operations on data. * (3) A function being performed or waiting to be performed. * (4) A program in operation. For example, a daemon is a system process that is always running on the system.

R

remote host. *See foreign host.*

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

S

server. (1) A function that provides services for users. A machine may run client and server processes at the same time. (2) A machine that provides resources to the network. It provides a network service, such as disk storage and file transfer, or a program that uses such a service.

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

System Administrator. The user who is responsible for setting up, modifying, and maintaining the Parallel Environment.

U

user. Anyone who requires the services of a computing system.

W

working directory. A collection of files to be manipulated by an FTP operation.

X

X. See the X window system.

X Window System. The UNIX industry's graphics windowing standard that provides simultaneous views of several executing programs or processes on high resolution graphics displays.

Index

A

AIX profilers
  using to profile parallel programs 91—92

D

debugging parallel programs
  with pdbx 56—71
  with xpdbx 72—89

E

execution environment setup 10—25

G

gprof, using to profile parallel programs 91—92

H

host list files, creating 16

L

LoadLeveler, submitting a POE job to 39

M

monitoring processor nodes using the System Status Array 50—52
monitoring processor nodes using VT 120—124
  adjusting sampling interval when 123
  selecting nodes for monitoring 122
  starting monitoring 122
mpcc shell script 8
mprcp shell script 9
mpxlf shell script 8

N

National Language Support (NLS) for PE xi

O

online help, accessing xi

P

Parallel Desktop 41—45
Parallel Environment (PE), overview 1—5
parallel programs
  compiling 8
  compiling for VT 107
parallel programs *(continued)*
  controlling program execution 32—38
  debugging 53—89
  executing 7—31
  monitoring execution using the Program Marker Array 46
  monitoring program execution 32—38
  profiling 91—92
  visualizing performance of 105—119
pdbx debugger
  controlling program execution using 69
  deleting breakpoints using 69
  deleting tracepoints using 69
  grouping tasks using 59
  setting breakpoints using 65
  setting command context using 59
  setting tracepoints using 66
  starting 53—56
  stopping 71
  using 56—71
pdbx subcommands 179—193
  active subcommand 59, 179
  alias subcommand 63, 179
  creating, removing, and listing aliases for 63
  dbx subcommand 181
  delete subcommand 69, 181
  dhelp subcommand 59, 182
  group subcommand 60—62, 182
  help subcommand 58, 184
  hook subcommand 70, 185
  load subcommand 64, 186
  obtaining information with 58
  on subcommand 59, 62, 186
  overview of 56
  quick reference listing 57
  quit subcommand 71, 188
  source subcommand 188
  status subcommand 68, 189
  stop subcommand 65, 190
  trace subcommand 66, 191
  unalias subcommand 63, 192
  unhook subcommand 70, 192
PE commands
  manual pages 165—212
  mpcc 166, 169
  mprcp 172
  mpxlf 173
  pdbx 176—193
  pmarray 194
  poe 196—204
  poestat 205
  vt 207

© Copyright IBM Corp. 1993

PE commands *(continued)*
xpdbx 210
performance monitoring 120—124
adjusting sampling interval during 123
selecting nodes for monitoring 122
starting monitoring 122
POE
commands file, loading nodes individually using 28
commands file, reading job steps from 30
controlling program execution using 32—38
controlling using Parallel Desktop 41—45
executing non-parallel programs using 31
executing parallel programs using 7—31
invoking executables in 25—31
monitoring program execution using 32—38
setting up execution environment 10—25
POE command-line flags 213—218
-cmdfile 28, 30
-euidevelop 32
-euidevice 23
-euilib 22
-euilibpath 23
-hostfile and -hfile 21
-infolevel and -ilevel 37
-labelio 37
-newjob 29
-pgmmodel 26
-pmdlog 38
-procs 15
-resd 21
-retry 33
-rmpool 24
-samplefreq and -sfreq 109
-stdinmode 34
-stdoutmode 35
-tbuffsize and -tbsize 109, 111
-tbuffwrap and tbwrap 111
-tmpdir 110
-tpermsize and -tpsize 109, 111
-tracedir and -tdir 110
-tracefile and -tfile 109
-tracelevel and -tlevel 108
-ttempsize and -ttsize 109, 111
changing the sampling interval for AIX kernel statistics using 109
generating diagnostic logs using 38
labeling task output using 37
maintaining partition for multiple job steps using 29
making POE wait for available nodes using 33
managing standard input using 34
managing standard output using 35
overriding environment variables using x
setting number of task processes using 15
setting the message reporting level using 37
specifying a commands file using 28, 30
specifying a directory for final trace output using 110
POE command-line flags *(continued)*
specifying a host list file using 21
specifying adapter set for message passing using 23
specifying additional error checking using 32
specifying CSS library implementation using 22
specifying node allocation through Resource Manager using 21
specifying programming model using 26
specifying temporary trace file size using 111
specifying the maximum size of output trace files using 111
specifying trace buffer size using 111
specifying trace file names using 109
specifying wraparound trace storage using 111
turning tracing on using 108
writing a trace file to a specified directory using 110
POE environment variables 213—218
changing the sampling interval for AIX kernel statistics using 109
generating diagnostic logs using 38
labeling task output using 37
maintaining partition for multiple job steps using 29
making POE ignore arguments using 33
making POE wait for available nodes using 33
managing standard input using 34
managing standard output using 35
managing storage for trace files using 109
MP_CMDFILE 28, 30
MP_EUIDEVELOP 32
MP_EUIDEVICE 23
MP_EUILIB 22
MP_EUILIBPATH 23
MP_FENCE 33
MP_HOSTFILE 21
MP_INFOLEVEL 37
MP_LABELIO 37
MP_NEWJOB 29
MP_NOARGLIST 33
MP_PERMSIZE 109
MP_PGMMODEL 26
MP_PMDLOG 38
MP_PROCS 15
MP_RESD 21
MP_RETRY 33
MP_RMPOOL 24
MP_SAMPLEFREQ 109
MP_STDINMODE 34
MP_STDOUTMODE 35
MP_TBUFFSIZE 109, 111
MP_TBUFFWRAP 111
MP_TMPDIR 110
MP_TPERMSIZE 111
MP_TRACEDIR 110
MP_TRACEFILE 109
MP_TRACELEVEL 108

POE environment variables *(continued)*
MP_TTEMPSIZE 109, 111
setting x
setting number of task processes using 15
setting the message reporting level using 37
specifying a commands file using 28, 30
specifying a directory for final trace output using 110
specifying a host list file 21
specifying adapter set for message passing using 23
specifying additional error checking using 32
specifying CSS library implementation using 22
specifying node allocation through Resource Manager using 21
specifying programming model using 26
specifying temporary trace file size using 111
specifying the maximum size of output trace files using 111
specifying trace buffer size using 111
specifying trace file names using 109
specifying wraparound trace storage using 111
turning tracing on using 108
writing a trace file to a specified directory using 110
prof, using to profile parallel programs 91—92
profiling parallel programs 91—92
Program Marker Array 46
displaying details of light on 48
displaying task output on 49
Parallel Utility Functions for 47
resource variables 220
setting the number of lights on 47
starting 48

T

trace files
adjusting playback speed of 118
cycling playback through 117
generating 106—111
loading for playback 114
returning playback to an earlier point in 116
starting playback of 115
stepping playback of 117
using VT to play back 112—119
trace records 105
trace visualization 105—119
generating trace files for 106—111
types of trace records for 105

V

Visualization Tool (VT) 93—163
closing views 98
opening views 98
quick operation of 96
Visualization Tool (VT) *(continued)*
resource variables 220—225
saving screen configurations for 103
starting 97
using for performance monitoring 120—124
using for trace visualization 105—119
view categories described 99
view descriptions 124—163
VT views
adjusting display rate for 101
adjusting magnification of 119
categories described 99
changing colors displayed on 101
closing 98
Communication/Program views 124—135
Computation views 135—146
Connectivity Graph 125
Context Switches (Bar Chart) 156
Context Switches (Graph) 157
descriptions of 124—163
Disk Reads (Bar Chart) 146
Disk Reads (Graph) 147
Disk Transfers (Bar Chart) 148
Disk Transfers (Graph) 149
Disk Views 146—152
Disk Writes (Bar Chart) 150
Disk Writes (Graph) 151
Interprocessor Communication 126
Kernel Utilization (Bar Chart) 135
Kernel Utilization (Graph) 136
Message Status Matrix 129
Network views 152—156
opening 98
Packets Received (Bar Chart) 152
Packets Received (Graph) 153
Packets Sent (Bar Chart) 154
Packets Sent (Graph) 155
Page Faults (Bar Chart) 158
Page Faults (Graph) 159
Processor Idle (Bar Chart) 140
Processor Idle (Graph) 141
Processor Utilization (3D Bar Chart) 144
Processor Wait (Bar Chart) 138
Processor Wait (Graph) 139
scrolling over history buffers 118
Source Code 131
System Calls (Bar Chart) 160
System Calls (Graph) 161
System Summary 162
System views 156—163
User Load Balance 145
User Utilization (Bar Chart) 142
User Utilization (Graph) 143

X

xpdbx debugger
changing the source code displayed by 88
controlling program execution with 77—82
debugging using multiple views 89
examining program data 85
examining program data with 82
grouping tasks using 73—75
hiding a task's break/trace information 85
hiding a task's data information 85
hiding a task's stack information 85
loading the partition using 75
setting breakpoints using 79
setting command context using 73
starting 53—56
stepping program execution with 81
tracing program execution using 86
using 72—89

Readers' Comments — We'd Like to Hear from You

IBM AIX Parallel Environment
Operation and Use
Release 2.0

Publication No. SH26-7230-01

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

Use this form to tell us what you think about this manual. If you prefer, you may send comments electronically to your IBM branch office, **Attention: Dept. 52QA-911, Kingston, NY**. Or you may fax your comments to us directly at: **FAX (914) 385-0662**.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

Name _____ Address _____

Company or Organization _____

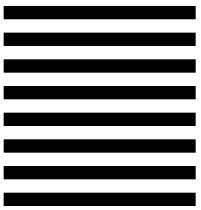
Phone No. _____

Readers' Comments — We'd Like to Hear from You
SH26-7230-01

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 52QA, Mail Station 911
1 Enterprise Drive
Kingston, NY 12401-1099

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in U.S.A.

SH26-7230-01