IBM Parallel Environment for AIX

Operation and Use, Volume 2, Part 1
Debugging and Visualizing

Version 2 Release 4

SC28-1980-02

IBM Parallel Environment for AIX

Operation and Use, Volume 2, Part 1 Debugging and Visualizing

Version 2 Release 4

SC28-1980-02

Note!

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

Third Edition (October 1998)

This edition applies to Version 2, Release 4 , Modification 0 of the IBM Parallel Environment for AIX (5765-543), and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address 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 55JA, Mail Station P384

522 South Road

Poughkeepsie, NY 12601-5400

United States of America

FAX (United States and Canada: 1+914+432-9405

FAX (Other Countries)

Your International Access Code)+1+914+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
IBM Mail Exchange: USIB6TC9 at IBMMAIL

Internet e-mail: mhvrcf@vnet.ibm.com

World Wide Web: http://www.rs6000.ibm.com (select Parallel Computing)

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

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

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

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

Contents

Notices . . . vii

Trademarks . . . ix

About This Book . . . xi
Who Should Use This Book . . . xi
How This Book is Organized . . . xi
  Overview of Contents . . . xi
  Typographic Conventions . . . xii
Related Publications . . . xii
  IBM Parallel Environment for AIX Publications . . . xiii
  Related IBM Publications . . . xiii
  Related Non-IBM Publications . . . xiv
National Language Support . . . xiv
Accessing Online Information . . . xiv
| Online Information Resources . . . xv
| Getting the Books Online . . . xv

Chapter 1. Using the pdbx Debugger . . . 1
pdbx Subcommands . . . 1
Starting the pdbx Debugger . . . 4
  Normal Mode . . . 4
  Attach Mode . . . 6
  Attach Screen . . . 7
Loading the Partition with the Load Subcommand . . . 10
  Displaying Tasks and their States . . . 11
  Grouping Tasks . . . 11
  Controlling Program Execution . . . 20
  Examining Program Data . . . 27
  Other Key Features . . . 33
  Other Important Notes on pdbx . . . 37
  Exiting pdbx . . . 39

Chapter 2. Using the pedb Debugger . . . 41
Starting the pedb Debugger . . . 41
  Normal Mode . . . 42
  Attach Mode . . . 43
  Attach Window . . . 44
The pedb Main Window . . . 46
Loading the Partition from the Load Executables Window . . . 48
  Program Search Path . . . 48
  The pedb Window with a Partition Loaded . . . 50

Chapter 3. Visualizing Program and System Performance . . . 109
Starting the Visualization Tool . . . 113
Opening and Closing Views . . . 114
Using VT for Trace Visualization . . . 117
  Types of Trace Records . . . 117
  Trace Record Timestamps . . . 118
  Generating Trace Files . . . 119

Using VT to Play Trace Files . . . 125
Using VT for Performance Monitoring . . . 139
The Performance Monitor Window . . . 140
Adjusting a View's Time Resolution and Colors . . . 144
Adjusting View Colors . . . 146
Saving and Loading a VT Configuration File . . . 147
View Descriptions . . . 148
View Types . . . 149
Communication/Program Views . . . 149
Computation Views . . . 162
Disk Views . . . 174
Network Views . . . 180
System Views . . . 184

Appendix A. Parallel Environment Tools Commands . . . 193
pdbx . . . 193
pdbx alias Subcommand . . . 198
pdbx assign Subcommand . . . 199
pdbx attach Subcommand . . . 200
pdbx attribute Subcommand . . . 200
pdbx back Subcommand . . . 201
pdbx call Subcommand . . . 201
pdbx case Subcommand . . . 201
pdbx catch Subcommand . . . 202
pdbx condition Subcommand . . . 202
pdbx cont Subcommand . . . 203
pdbx dbx Subcommand . . . 203
pdbx delete Subcommand . . . 204
pdbx detach Subcommand . . . 205
pdbx dhelp Subcommand . . . 205
pdbx display memory Subcommand . . . 206
pdbx down Subcommand . . . 207
pdbx dump Subcommand . . . 207
pdbx file Subcommand . . . 207
pdbx func Subcommand . . . 208
pdbx goto Subcommand . . . 208
pdbx gotoi Subcommand . . . 208
pdbx group Subcommand . . . 208
pdbx halt Subcommand . . . 210
pdbx help Subcommand . . . 211
pdbx hook Subcommand . . . 211
pdbx ignore Subcommand . . . 212
pdbx list Subcommand . . . 213
pdbx listi Subcommand . . . 214
pdbx load Subcommand . . . 215
pdbx map Subcommand . . . 215
pdbx mutex Subcommand . . . 215
pdbx next Subcommand . . . 216
pdbx nexti Subcommand . . . 217
pdbx on Subcommand . . . 217
pdbx print Subcommand . . . 219
pdbx quit Subcommand . . . 220
pdbx registers Subcommand . . . 220
pdbx return Subcommand . . . 220

pdbx search Subcommand 221
pdbx set Subcommand 221
pdbx sh Subcommand 221
pdbx skip Subcommand 222
pdbx source Subcommand 222
pdbx status Subcommand 222
pdbx step Subcommand 223
pdbx stepi Subcommand 224
pdbx stop Subcommand 224
pdbx tasks Subcommand 226
pdbx thread Subcommand 226
pdbx trace Subcommand 228
pdbx unalias Subcommand 229
pdbx unhook Subcommand 230
pdbx unset Subcommand 231
pdbx up Subcommand 231
pdbx use Subcommand 231
pdbx whatis Subcommand 232
pdbx where Subcommand 232
pdbx whereis Subcommand 232
pdbx which Subcommand 232
pedb 233
vt 235

Appendix B. Command Line Flags for Normal or Attach Mode 239

Appendix C. Exporting Arrays to Hierarchical Data Format (HDF) 241

Appendix D. Visualization Customization and Data Explorer Samples . 243

Appendix E. Customizing Tool Resources 247

Glossary of Terms and Abbreviations 249

Index 257

Notices

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

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

IBM Director of Licensing

IBM Corporation

500 Columbus Avenue

Thornwood, NY 10594

USA

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

IBM Corporation

Mail Station P300

522 South Road

Poughkeepsie, NY 12601-5400

USA

Attention: Information Request

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

Trademarks

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

AIX
AIX/6000
IBM
LoadLeveler
Micro Channel
RISC System/6000
RS/6000
POWERparallel
SP

Microsoft, Windows, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation.

PostScript is a trademark of Adobe Systems, Incorporated.

Motif is a trademark of Open Software Foundation.

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

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

About This Book

This book describes the facilities and tools for the IBM Parallel Environment (PE) for AIX program product and how to use them to debug and analyze parallel programs. It describes the various PE tools for debugging parallel programs and visualizing a program's 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 *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*, (GC23-3894) and *IBM Parallel Environment for AIX: MPL Programming and Subroutine Reference* (GC23-3893).

This book assumes that AIX Version 4.3.2 or later , X-Windows**, and the PE software are already installed. It also assumes that you have been authorized to run the Parallel Operating Environment (POE). The PE software is designed to run on an IBM RS/6000 SP, an RS/6000 network cluster, or on a mixed system where additional RS/6000 processors supplement an SP system. For complete information on installing the PE software and setting up users, see *IBM Parallel Environment for AIX: Installation*, (GC23-3892). Also, see the appropriate AIX 4.3.2 or later documentation listed under "Related Publications" on page xii. For information on POE and executing parallel programs, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* and *IBM Parallel Environment for AIX: Hitchhiker's Guide*

Who Should Use This Book

This book is designed primarily for end users and application developers. It is also intended for those who run 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 This Book is Organized

Overview of Contents

This book contains the following information:

- Chapter 1, "Using the pdbx Debugger" on page 1 describes the Parallel Environment's command line debugger – **pdbx**. This tool uses a line-oriented interface, allowing you to invoke a parallel program from an ASCII terminal.
- Chapter 2, "Using the pedb Debugger" on page 41 describes the other Parallel Environment debugger – **pedb**. This tool uses an X-windows interface for interactive debugging.
- Chapter 3, "Visualizing Program and System Performance" on page 109 describes the PE Visualization Tool (VT). Intended for application developers, 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 Tools Commands” on page 193 contains the manual pages for the PE commands discussed throughout this book.
- Appendix B, “Command Line Flags for Normal or Attach Mode” on page 239 shows the command line flags for **pedb** debugging in normal or attach mode.
- Appendix C, “Exporting Arrays to Hierarchical Data Format (HDF)” on page 241 describes additional information about the format of the data used for the **pedb** export feature.
- Appendix D, “Visualization Customization and Data Explorer Samples” on page 243 shows additional information on the Data Explorer samples that are included as a set of pre-packaged interfaces for the visualization of program data.
- Appendix E, “Customizing Tool Resources” on page 247 describes how to customize X-Windows resources for PE tools.

Typographic Conventions

This book uses the following typographic conventions:

Type Style	Used For
bold	**Bold** words or characters represent system elements that you must use literally, such as command names, flag names, and path names. **Bold** words also indicate the first use of a term included in the glossary.
italic	*Italic* words or characters represent variable values that you must supply. *Italics* are also used for book titles and for general emphasis in text.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.

In addition to the highlighting conventions, this manual uses the following conventions when describing how to perform tasks. User actions appear in uppercase boldface type. For example, if the action is to enter the **pedb** command, this manual presents the instruction as:

ENTER **pedb**

The symbol “●” indicates the system response to an action. So the system's response to entering the **pedb** command would read:

● The **pedb** main window opens.

Related Publications

IBM Parallel Environment for AIX Publications

- *IBM Parallel Environment for AIX: General Information*, GC23-3906
- *IBM Parallel Environment for AIX: Hitchhiker's Guide*, GC23-3895
- *IBM Parallel Environment for AIX: Installation*, GC28-1981
- *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*, SC28-1979
- *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*, GC23-3894
- *IBM Parallel Environment for AIX: MPL Programming and Subroutine Reference*, GC23-3893
- *IBM Parallel Environment for AIX: Messages*, GC28-1982
- *IBM Parallel Environment for AIX: Licensed Program Specification*, GC23-3896

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

Related IBM Publications

- *IBM AIX Version 4 Getting Started*, SC23-2527
- *IBM AIX General Concepts and Procedures for RS/6000* GC23-2202
- *IBM AIX Version 4 Files Reference*, SC23-2512
- *IBM AIX Version 4 System Management Guide: Communications and Networks*, SC23-2526
- *IBM AIX Version 4.1 Installation Guide* SC23-2550
- *IBM AIX Version 4.2 Installation Guide* SC23-1924
- *IBM AIX Version 4 Commands Reference*, SBOF-1851 (all volumes)
- *IBM AIX Versions 3.2 and 4 Performance Tuning Guide* SC23-2365
- *IBM AIX Version 4 Messages Guide and Reference* SC23-2641
- *IBM AIX Version 4.1 Network Installation Management Guide and Reference*, SC23-2627
- *IBM AIX Version 4.2 Network Installation Management Guide and Reference*, SC23-1926
- *IBM AIX Version 4 System Management Guide: Operating System and Devices*, SC23-2525
- *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*, SC23-2533
- *IBM AIX Version 4 Communications Programming Concepts* SC23-2610
- *Diskless Workstation Management Guide*, SC23-2433
- *C++ for AIX/6000: Language Reference*, SC09-1606
- *C++ for AIX/6000: Standard Class Library Reference*, SC09-1604
- *C++ for AIX/6000: User's Guide*, SC09-1605
- *IBM Performance Toolbox 1.2 and 2 for AIX: Guide and Reference*, SC23-2625

Related Non-IBM Publications

- Almasi, G., Gottlieb, A., *Highly Parallel Computing* Benjamin-Cummings Publishing Company, Inc., 1989.
- Gropp, W., Lusk, E., Skjellum, A., *Using MPI*, The MIT Press, 1994.
- Message Passing Interface Forum, MPI: *A Message-Passing Interface Standard* Version 1.1, University of Tennessee, Knoxville, Tennessee, June 6, 1995.
- Foster, I., *Designing and Building Parallel Programs* Addison-Wesley, 1995.
- Pfister, Gregory, F., *In Search of Clusters* Prentice Hall, 1995.

National Language Support

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

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

export LANG=C

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

/usr/lib/nls/msg/C
/usr/lib/nls/msg/En_US
/usr/lib/nls/msg/en_US

If your site is using its own translations of the message catalogs, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For additional information on NLS and message catalogs, see *IBM Parallel Environment for AIX: Messages* and *AIX for RS/6000: General Programming Concepts*.

Accessing Online Information

In order to use the PE man pages or access the PE online (HTML) publications, the **ppe.pedocs** file set must first be installed. To view the PE online publications, you also need access to an HTML document browser such as Netscape. An index to the HTML files that are provided with the **ppe.pedocs** file set is installed in the **/usr/lpp/ppe.pedocs/html** directory.

Online Information Resources

If you have a question about the SP, PSSP, or a related product, the following online information resources make it easy to find the information:

- Access the new SP Resource Center by issuing the command: **/usr/lpp/ssp/bin/resource_center**. Note that the **ssp.resctr** fileset must be installed before you can do this.

  If you have the Resource Center on CD ROM, see the readme.txt file for information on how to run it.

- Access the RS/6000 Web Site at: **http://www.rs6000.ibm.com**.

Getting the Books Online

All of the PE books are available in Portable Document Format (PDF). They are included on the product media (tape or CD ROM), and are part of the **ppe.pedocs** file set. If you have a question about the location of the PE softcopy books, see your System Administrator.

To view the PE PDF publications, you need access to the Adobe Acrobat Reader 3.0.1. The Acrobat Reader is shipped with the AIX Version 4.3 Bonus Pack and is also freely available for downloading from the Adobe web site at URL **http://www.adobe.com**.

As stated above, you can also view or download the PE books from the IBM RS/6000 web site at **http://www.rs6000.ibm.com**. At the time this manual was published, the full path was **http://www.rs6000.ibm.com/resource/aix_resource/sp_books.** However, note that the structure of the RS/6000 web site can change over time.

Chapter 1. Using the pdbx Debugger

This chapter describes the **pdbx** debugger. This debugger extends the **dbx** debugger's line-oriented interface and subcommands. Some of these subcommands, however, have been modified for use on parallel programs. The **pdbx** debugger is a POE application with some modifications on the *home node* to provide a user interface.

Before invoking a parallel program using **pdbx** for interactive debugging, you first need to compile the program and set up the execution environment. See *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for more information on the following:

- Compiling the program. Be sure to specify the **-g** flag when compiling the program. This produces an object file with symbol table references needed for symbolic debugging. It is also advisable to not use the optimization option, **-O**. Using the debugger on optimized code may produce inconsistent and erroneous results. For more information on the **-g** and **-O** compiler options, refer to their use on other compiler commands such as **cc** and **xlf**. These compiler commands are described in *IBM AIX Version 4 Commands Reference* or your online manual pages.
- Copying files to individual nodes. Like **poe**, **pdbx** requires that your application program be available to run on each node in your partition. To support source level debugging, **pdbx** requires the source files to be available as well. You will generally use the same mechanism to make the source files accessible as you used for the application program.
- Setting up the execution environment.

As you read these steps, keep in mind that **pdbx** accepts almost all the option flags that **poe** accepts, and responds to the same environment variables.

Also, throughout this book, keep in mind the following information.

The RS/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 network, so the parallel tasks of your partition can communicate to exchange data or synchronize execution.

pdbx Subcommands

Table 1 on page 2 outlines the **pdbx** subcommands described in this chapter. Complete syntax information for all these subcommands is also provided under the entry for the **pdbx** command in Appendix A, “Parallel Environment Tools Commands” on page 193.

The debugger supports most of the familiar **dbx** subcommands, as well as some additional **pdbx** subcommands. In **pdbx**, *command context* refers to a setting that controls which task(s) receive the subcommands entered at the **pdbx** command prompt.

pdbx subcommands can either be *context sensitive* or *context insensitive*. The debugger directs context sensitive subcommands to just the tasks in the current command context. Command context has no bearing on context insensitive commands, which control overall debugger behavior, and are generally processed on the home node only. These include subcommands for setting help and other information, and ending a **pdbx** session.

You can set the command context on a single task or a group of tasks as described in “Setting Command Context” on page 15.

Table 1. pdbx Subcommands

Context Insensitive pdbx Subcommands

This subcommand:	**Is used to:**	**For more information see:**
alias [alias_name string]	Set or display aliases.	“Creating, Removing, and Listing Command Aliases” on page 34
attach <[all | task_list]>	Attach the debugger to some or all the tasks of a given **poe** job.	“Attach Mode” on page 6
detach	Detach **pdbx** from all tasks that were attached. This subcommand causes the debugger to exit but leaves the **poe** application running.	“Exiting pdbx” on page 39
dhelp [dbx_command]	Display a brief list of **dbx** commands or help information about them.	“Accessing Help for dbx Subcommands” on page 33
group <action> [group_name] [task_list]	Manipulate groups. The actions are **add**, **change**, **delete**, and **list**. To indicate a range of tasks, enter the first and last task numbers, separated by a colon or dash. To indicate individual tasks, enter the numbers, separated by a space or comma.	“Grouping Tasks” on page 11
help [subject]	Display a list of **pdbx** commands and topics or help information about them.	“Accessing Help for pdbx Subcommands” on page 33
on <[group | task]> [command]	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.	“Setting the Current Command Context” on page 15
quit	End a **pdbx** session.	“Exiting pdbx” on page 39
source <cmd_file>	Execute **pdbx** subcommands from a specified file. **Note:** The file may contain context sensitive commands.	“Reading Subcommands From a Command File” on page 36
tasks [long]	Display information about all the tasks in the partition.	“Displaying Tasks and their States” on page 11
unalias alias_name	Remove a command alias specified by the **alias** subcommand.	“Creating, Removing, and Listing Command Aliases” on page 34

Context Sensitive pdbx Subcommands

This Subcommand:	Is used to:	For more information see:
delete <[event_list | * | all]>	Remove breakpoints and tracepoints set by the **stop** and **trace** subcommands. To indicate a range of events, enter the first and last event numbers, separated by a colon or a dash. To indicate individual events, enter the number(s), separated by a space or comma.	"Deleting pdbx Events" on page 25
dbx <dbx_command>	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. **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**, **screen**, and the **sh** subcommand with no arguments.	the online PE manual page for **pdbx**. This manual page also appears in Appendix A, "Parallel Environment Tools Commands" on page 193.
hook	Regain control over an unhooked task.	"Unhooking and Hooking Tasks" on page 26
list [line_number | line_number, line_number | procedure]	Display lines of the current source file, or of a procedure.	"Displaying Source" on page 32
load <program> [program_arguments]	Load a program on each node in the current context. This can only be issued once per task per **pdbx** session. **pdbx** will look for the program in the current directory unless a relative or absolute pathname is specified.	"Loading the Partition with the Load Subcommand" on page 10
print <[expression | procedure]>	Print the value of an expression, or run a procedure and print the return code of that procedure.	"Viewing Program Variables" on page 28
status [all]	Display a list of breakpoints and tracepoints set by the **stop** and **trace** subcommands in the current context. If "all" is specified, all events, regardless of context are shown.	"Checking Event Status" on page 26
stop	Set a breakpoint for tasks in the current context. Breakpoints are stopping places in your program that halt execution.	"Setting Breakpoints" on page 20
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.	"Setting Tracepoints" on page 22
unhook	Unhook a task or group of tasks. Unhooking allows the task(s) to run without intervention from the debugger.	"Unhooking and Hooking Tasks" on page 26
where	Display a list of active procedures and functions.	"Viewing Program Call Stacks" on page 27

<**Ctrl-c**>	Regain debugger control when some tasks in the current context are running. This causes a **pdbx** subset prompt to be displayed, which allows a subset of the **pdbx** function to be performed.	“Context Switch when Blocked” on page 17

Starting the pdbx Debugger

You can start the **pdbx** debugger in either *normal* mode or *attach* mode. In normal mode your program runs under the control of the debugger. In attach mode you attach to a program that is already running. Certain options and functions are only available in one of the two modes. Since **pdbx** is a source code debugger, some files need to be compiled with the **-g** option so that the compiler provides debug symbols, source line numbers, and data type information.

When the application is started using **pdbx** in normal mode, debugger control of the application is given to the user by default at the first executable source line within the main routine. This is function *main* in C code or the the routine defined by the *program* statement in Fortran. In Fortran, if there is no *program* statement, the program name defaults to *main*. If the file containing the main routine is not compiled with **-g** the debugger will exit. The environment variable **MP_DEBUG_INITIAL_STOP** can be set before starting the debugger to manually set an alternate file name and source line where the user initially receives debugger control of the application. Refer to the appendix on POE environment variables and command-line flags in *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*

Normal Mode

The way you start the debugger in normal mode depends on whether the program(s) you are debugging follow the SPMD (Single Program Multiple Data) or MPMD (Multiple Program Multiple Data) model of parallel programming. In the SPMD model, the same program runs on each of the nodes in your partition. In the MPMD model, different programs can run on the nodes of your partition.

If you are debugging an SPMD program, you can enter its name on the **pdbx** command line. It will be loaded on all the nodes of your partition automatically. If you are debugging an MPMD program, you will load the tasks of your partition after the debugger is started. **pdbx** will look for the program in the current directory unless a relative or absolute pathname is specified.

ENTER **pdbx** [*program* [*program_options*]] [*poe opt ions*] [**-c** *command_file*] [**-d** *nesting_depth*] [**-I** *directory* [**-I** *directory*]...] [**-F**] [**-x**]

- This starts **pdbx**. If you specified a *program*, it is loaded on each node of your partition and you see the message:

```
0031-504  Partition loaded ...
```

You will then see the pdbx prompt:

```
pdbx(all)
```

The prompt shows the command context `all`. For more information see “Setting Command Context” on page 15.

ENTER **pdbx -a** *poe process id* [*limited poe options*] [**-c** *command_file*] [**-d** *nesting_depth*] [**-I** *directory* [**-I** *directory*]...] [**-F**] [**-x**]

- This starts **pdbx** in attach mode. See "Attach Mode" on page 6 for more information.

ENTER **pdbx -h**

- This writes the **pdbx** usage to STDERR. It includes **pdbx** command line syntax and a description of **pdbx** options.

The options you specify with the **pdbx** command can be program options, POE options, or **pdbx** options listed in Table 2. Program options are those that your application program will understand.

You can use the same command-line flags on the **pdbx** command 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 *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*

Note: **poe** uses the **PATH** environment variable to find the program, while **pdbx** does not.

After **pdbx** initializes, the **pdbx** command prompt displays to indicate that **pdbx** is ready for a command.

Table 2 (Page 1 of 2). Debugger Option Flags (pdbx)

Use this flag:	To:	For example:
-a	Attach to a running **poe** job by specifying its process id. This must be executed from the node where the **poe** job was initiated. When using the debugger in attach mode there are some debugger command line arguments that should not be used. In general, any arguments that control how the partition is set up or specify application names and arguments should not be used.	To attach the **pdbx** debugger to an already running **poe** job. **ENTER** **pdbx -a** <*poe_process_id*>
-c	Read **pdbx** startup commands from the specified *commands_file*. 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 Command File" on page 36 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*
-d	Set 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*

Table 2 (Page 2 of 2). Debugger Option Flags (pdbx)

Use this flag:	To:	For example:
-F	This flag can be used to turn off *lazy reading* mode. Turning lazy reading mode off forces the remote **dbx** sessions to read all symbol table information at startup time. By default, lazy reading mode is on. Lazy reading mode is useful when debugging large executable files, or when paging space is low. With lazy reading mode on, only the required symbol table information is read upon initialization of the remote **dbx** sessions. Because all symbol table information is not read at **dbx** startup time when in lazy reading mode, local variable and related type information will not be initially available for functions defined in other files. The effect of this can be seen with the **whereis** command, where instances of the specified local variable may not be found until the other files containing these instances are somehow referenced.	To start the **pdbx** debugger and read all symbol table information: **ENTER** **pdbx -F**
-h	Write the **pdbx** usage to STDERR then exit. This includes **pdbx** command line syntax and a description of **pdbx** options.	**ENTER** **pdbx -h**
-I (upper case i)	Specify a directory to be searched for an executable's source files. This flag must be specified multiple times to set multiple paths. (Once **pdbx** is running, this list can be overridden on a group or single node basis with the **use** command.)	To add *directory1* to the list of directories to be searched when starting the **pdbx** debugger: **ENTER** **pdbx -I** *dir1* 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** *dir1* **-I** *dir2*
-x	Prevent 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**

These **pdbx** flags are closely tied to the flags supported by **dbx**. For more information on the option flags described in this table, refer to their use with **dbx** as described in *IBM AIX Version 4 Commands Reference* and *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*

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 Tools Commands” on page 193.

Attach Mode

The **pdbx** debugger provides an attach feature, which allows you to attach the debugger to a parallel application that is currently executing. This feature is typically used to debug large, long running, or apparently “hung” applications. The debugger attaches to any subset of tasks without restarting the executing parallel program.

Parallel applications run on the partition managed by **poe**. For attach mode, you must specify the appropriate process identifier (PID) of the **poe** job, so the debugger can attach to the correct application processes contained in that particular job. To get the PID of the **poe** job, enter the following command on the node where **poe** was started:

```
$ ps -ef | grep poe
```

You initiate attach mode by invoking **pdbx** with the **-a** flag and the PID of the appropriate **poe** process:

```
$ pdbx -a <poe PID>
```

For example, if the process id of the **poe** process is 12345 then the command would be:

```
$ pdbx -a 12345
```

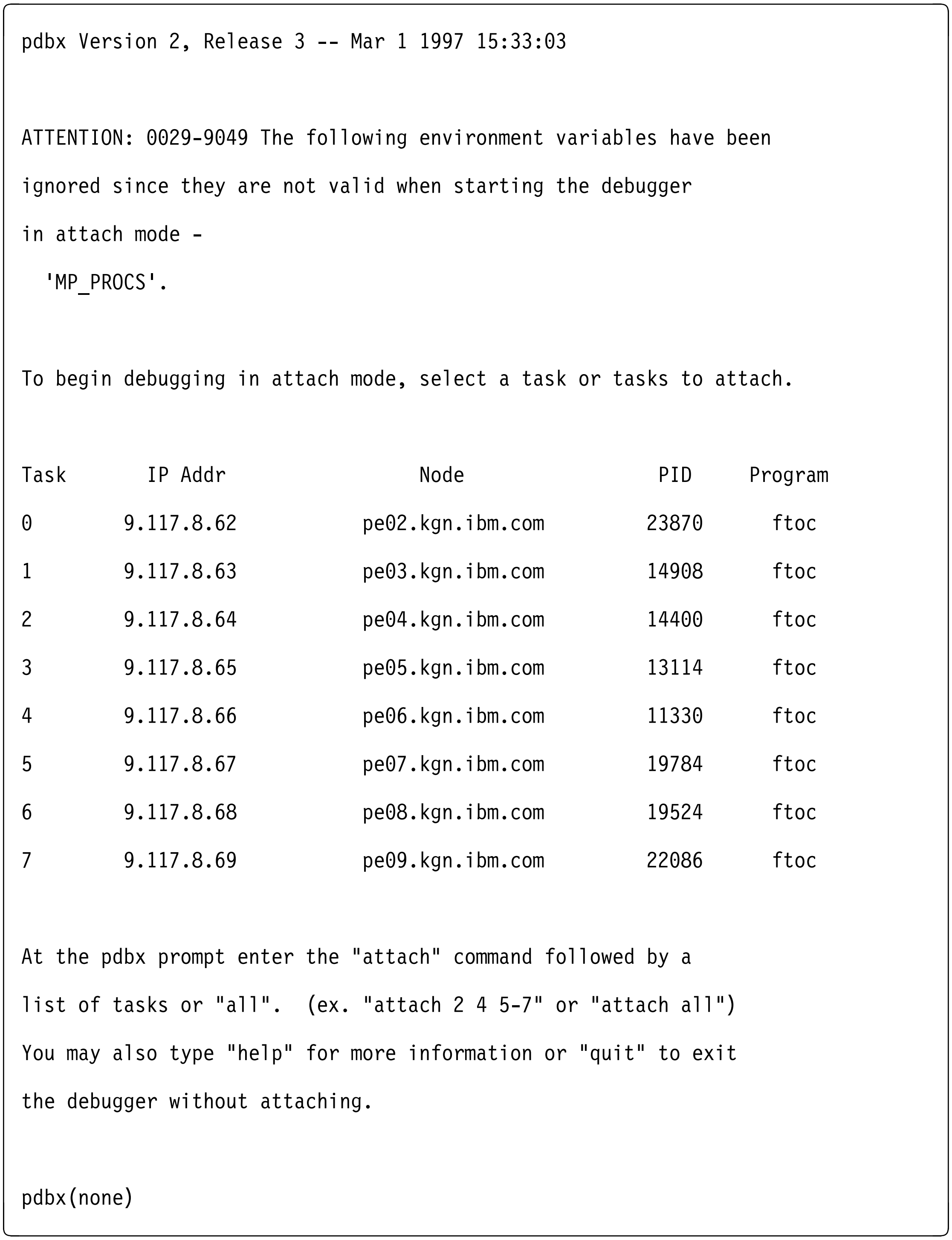
After you invoke the debugger in attach mode, it displays a list of tasks you can choose. The paging tool used to display the menu will default to **pg -e** unless another pager is specified by the **PAGER** environment variable.

pdbx starts by showing a list of task numbers that comprise the parallel job. The debugger obtains this information by reading a configuration file created by **poe** when it begins a job step. At this point you must choose a subset of that list to attach the debugger. Once you make a selection and the attach debug session starts, you cannot make additions or deletions to the set of tasks attached to. It is possible to attach a different set of tasks by detaching the debugger and attaching again, then selecting a different set of tasks.

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

Attach Screen

Figure 1 shows a sample **pdbx** Attach screen.



```
pdbx Version 2, Release 3 -- Mar 1 1997 15:33:03

ATTENTION: 0029-9049 The following environment variables have been
ignored since they are not valid when starting the debugger
in attach mode -
  'MP_PROCS'.

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

Task       IP Addr                 Node                 PID     Program
0        9.117.8.62           pe02.kgn.ibm.com         23870      ftoc
1        9.117.8.63           pe03.kgn.ibm.com         14908      ftoc
2        9.117.8.64           pe04.kgn.ibm.com         14400      ftoc
3        9.117.8.65           pe05.kgn.ibm.com         13114      ftoc
4        9.117.8.66           pe06.kgn.ibm.com         11330      ftoc
5        9.117.8.67           pe07.kgn.ibm.com         19784      ftoc
6        9.117.8.68           pe08.kgn.ibm.com         19524      ftoc
7        9.117.8.69           pe09.kgn.ibm.com         22086      ftoc

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

pdbx(none)
```

Figure 1. pdbx Attach screen

The **pdbx** Attach screen contains a list of tasks and, for each task, the following information:

- Task - the task number
- IP - the ip address of the node on which the task/application is running
- Node - the name of the node on which the task/application is running, if available

- PID - the process identifier of the task/application
- Program - the name of the application and arguments, if any.

After initiating attach mode, you can select a set of tasks to attach to. At the command prompt:

ENTER **attach all**

OR

ENTER **attach** followed by the *task_list* (see "Syntax for task_list" on page 12 for the correct syntax).

It is also possible to use the **quit** or **help** command at this prompt. Any other command will produce an error message. The **quit** command will not kill the application at this point, since the debugger has not been attached as of yet.

Note: When debugging in attach mode, the **load** subcommand is not available. An error message is displayed if an attempt is made to use it.

Other Compiling Options

pdbx provides substantial information when debugging an executable compiled with the **-g** option. However, you may find it useful to attach to an application not compiled with **-g**. **pdbx** allows you to attach to an application not compiled with **-g**, however, the information provided is limited to a stack trace.

You can also attach **pdbx** to an application compiled with both the **-g** and optimization flags. However, the optimized code can cause some confusion when debugging. For example, when stepping through code, you may notice the line marker points to different source lines than you would expect. The optimization causes this re-mapping of instructions to line numbers.

Command Line Arguments

You should not use certain command line arguments when debugging in attach mode. If you do, the debugger will not start, and you will receive a message saying the debugger will not start. In general, do not use any arguments that control how the debugger partition is set up or that specify application names and arguments. You do not need information about the application, since it is already running and the debugger uses the PID of the **poe** process to attach. Other information the debugger needs to set up its own partition, such as node names and PIDs, comes from the configuration file and the set of tasks you select. See Appendix B, "Command Line Flags for Normal or Attach Mode" on page 239 for a list of command line flags showing which ones are valid in normal and in attach debugging mode.

The information in the appendix is also true for the corresponding environment variables, however **pdbx** ignores the invalid setting. The debugger displays a message containing a list of the variables it ignores, and continues.

For example, if you had **MP_PROCS** set, when the debugger starts in attach mode it ignores the setting. It displays a message saying it ignored **MP_PROCS**, and continues initializing the debug session.

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 task of your partition. If not, you need to load your partition using the **load** subcommand. **pdbx** will look for the program in the current directory unless a relative or absolute pathname is specified. The Partition Manager allocates the tasks 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 tasks 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.

The following **pdbx** commands are available before the program is loaded on all tasks:

- alias
- group
- help
- load
- on
- quit
- source
- tasks
- unalias

To load the same executable on all tasks 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 tasks in the partition, and the message "Partition loaded..." displays. The parallel program runs up to the first executable statement and stops. **Note:** The example above has the same effect as putting the program name and options on the command line.	**SET** the command context before loading each program. For example, say your partition consists of five tasks numbered 0 through 4. To load a program named *program1* on task 0 and a program named *program2* on tasks 1 through 4, you would: **ENTER** **on** *0* ● The debugger sets the command context on task 0 **ENTER** **load** *program1* [program_options] The debugger loads *program1* on task 0. **ENTER** **group add** *groupa 1-4* ● The debugger creates a task group named *groupa* consisting of tasks 1 through 4. **ENTER** **on** *groupa* The debugger sets the command context on tasks 1 through 4. **ENTER** **load** *program2* [program_options] ● The debugger loads *program2* onto tasks 1 through 4, and the message "Partition loaded..." displays. The parallel program runs up to the first executable statement and stops.

Displaying Tasks and their States

With the **tasks** subcommand, you display information about all the tasks in the partition. Task state information is always displayed (see Table 3 on page 15 for information on task states). If you specify “long” after the command, it also displays the name, ip address, and job manager number associated with the task.

Following is an example of output produced by the **tasks** and **tasks long** command.

```
pdbx(others) tasks

  0:D     1:D     2:U     3:U     4:R     5:D     6:D     7:R



pdbx(others) tasks long

  0:Debug ready   pe04.kgn.ibm.com                 9.117.8.68       -1

  1:Debug ready   pe03.kgn.ibm.com                 9.117.8.39       -1

  2:Unhooked      pe02.kgn.ibm.com                 9.117.11.56      -1

  3:Unhooked      augustus.kgn.ibm.com             9.117.7.77       -1

  4:Running       pe04.kgn.ibm.com                 9.117.8.68       -1

  5:Debug ready   pe03.kgn.ibm.com                 9.117.8.39       -1

  6:Debug ready   pe02.kgn.ibm.com                 9.117.11.56      -1

  7:Running       augustus.kgn.ibm.com             9.117.7.77       -1
```

Grouping Tasks

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 group name you choose. None of these tasks need to have been loaded 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/worker program. Task 0 is the master task, controlling a number of worker 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.

Syntax for group_name

Provide a group name that is no longer than 32 characters which starts with an alphabetic character, and is followed by any alphanumeric character combination.

Syntax for task_list

To indicate a range of tasks, enter the first and last task numbers, separated by a colon or dash. To indicate individual tasks, enter the numbers, separated by a space or comma.

Note: Group names “all,” “none,” and “attached” are reserved group names. They are used by the debugger and cannot be used in the **group add** or **group delete** commands. However, the group “all” or “attached” can be renamed using the **group change** command, if it currently exists in the debugging session.

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 *task_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 task was added to group "groupa".`
a collection of tasks	tasks 6, 8, and 10	**group add** *groupa 6 8 10*	`3 tasks were added to group "groupa".`
a range of tasks	tasks 6 through 10	**group add** *groupa 6:10*	`5 tasks were added to group "groupa".`
a range of tasks	tasks 6 through 10	**group add** *groupa 6-10*	`5 tasks were 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 *task_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*	`Task: 6 was successfully`
			`deleted from group "groupa".`

a collection of tasks	task 6, 8, and 10	**group delete** *groupa 6 8 10*	`Task: 6 was successfully`
			`deleted from group "groupa".`
			`Task: 8 was successfully`
			`deleted from group "groupa".`
			`Task: 10 was successfully`
			`deleted from group "groupa".`

a range of tasks	tasks 6 through 10	**group delete** *groupa 6:10*	`Task: 6 was successfully`
			`deleted from group "groupa".`
			`Task: 7 was successfully`
			`deleted from group "groupa".`
			`Task: 8 was successfully`
			`deleted from group "groupa".`
			`Task: 9 was successfully`
			`deleted from group "groupa".`
			`Task: 10 was successfully`
			`deleted from group "groupa".`

a range of tasks	tasks 6 through 8	**group delete** *groupa 6-8*	`Task: 6 was successfully`
			`deleted from group "groupa".`
			`Task: 7 was successfully`
			`deleted from group "groupa".`
			`Task: 8 was successfully`
			`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:

```
Group "group1" has been renamed to "groupa".
```

Listing Task Groups

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

group list [*group_name*]

You can use the list action 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.
		`pdbx(0) group list`
		`allTasks      0:R     1:D     2:D     3:U     4:U     5:D     6:D`
		`              7:D     8:D     9:D    10:D    11:D`
		`evenTasks     0:R     2:D     4:U     6:D     8:D    10:R`
		`oddTasks      1:D     3:U     5:D     7:D     9:D    11:R`
		`master        0:R`
		`workers       1:D     2:D     3:U     4:U     5:D     6:D     7:D`
		`              8:D     9:D    10:R    11:R`

list all the members of a single task group	**group list** *oddTasks*	The debugger displays a list of all the members of task group *oddTasks*. `1:D    3:U    5:D    7:D    9:D    11:R`

When you list tasks, a single letter will follow each task number. The following table represents the state of affairs on the remote tasks. Common states are “debug ready,” where **pdbx** commands can be issued, and running, where the application has control and is executing.

Table 3. Task States

This letter displayed after a task number:	Represents:	And indicates that:
N	Not loaded	the remote task has not yet been loaded with an executable.
S	Starting	the remote task is being loaded with an executable.
D	Debug ready	the remote task is stopped and debug commands can be issued.
R	Running	the remote task is in control and executing the program.
X	Exited	the remote task has completed execution.
U	Unhooked	the remote task is executing without debugger intervention.
E	Error	the remote task is in an unknown state.

Figure 2 on page 15 illustrates the relationship between the **pdbx** debugger, which runs on the home task, and the various **dbx** processes running on the remote tasks. When thinking about "task states," consider the perspective of the remote tasks which are each running a copy of **dbx**. **pdbx** attempts to coordinate activities in multiple **dbx** sessions. There are times when this is not possible, typically when one or more tasks have not yet stopped. In this case, from a remote task's **dbx** perspective, a **dbx** prompt has not yet been displayed, and your application is still running. Similarly, **pdbx** will not display a **pdbx** prompt until all the remote **dbx** sessions are "debug ready."

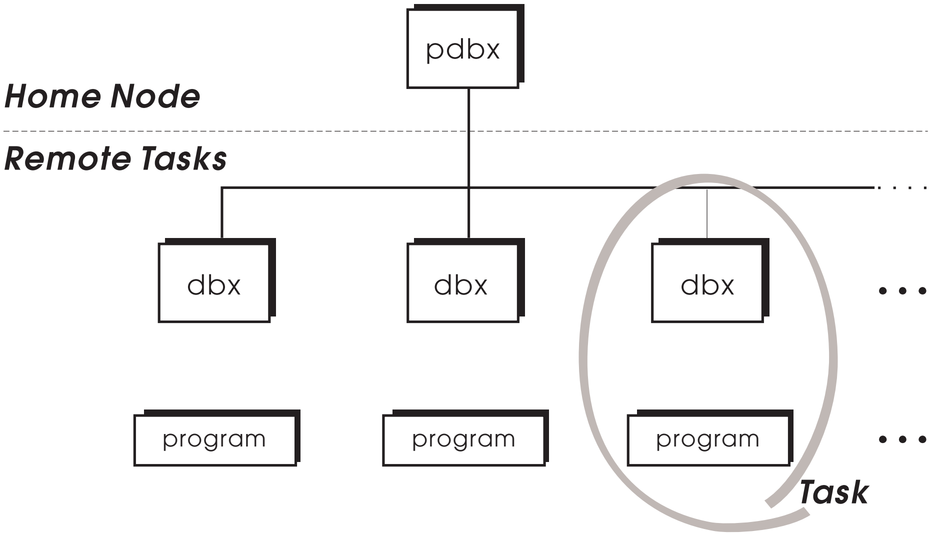


Figure 2. Relationship between home node (pdbx) and remote tasks (dbx processes)

Setting 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 use the **on** subcommand to set the current command context on one task or a group 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 will 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 and cannot be deleted. 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)
```

When you switch context using **on** *context_name*, and the new context has at least one task in the “running” state, a message is displayed stating that at least one task is in the “running” state. No **pdbx** prompt is displayed until all tasks in this context are in the “debug ready” state.

When you switch to a context where all tasks are in the “debug ready” state, the **pdbx** prompt is displayed immediately, indicating **pdbx** is ready for a command.

At the **pdbx** subset prompt, **on** *context_name* causes one of the following to happen: either a **pdbx** prompt is displayed; or a message is displayed indicating the reason why the **pdbx** prompt will be displayed at a later time. This is generally because one of the tasks is in “running” state. See “Context Switch when Blocked” on page 17 for more information.

Temporarily Deviating from the Current Command Context: There are times when it is convenient 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 **pdbx** prompt will be presented after all of the tasks in the temporary context have completed the command specified. It is possible, using <**Ctrl-c**> followed by the **back** or the **on** command, to issue further **pdbx** commands in the original context. 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 *groupa*. 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.

At a **pdbx** prompt, you cannot use **on** *context_name pdbx_command* if any of the tasks in the specified context are running.

Context Switch when Blocked

When a task is blocked (there is no **pdbx** prompt), you can press <**Ctrl-c**> to acquire control. This displays the **pdbx** subset prompt `pdbx-subset([group | task])`, and provides a subset of **pdbx** functionality including:

- Changing the current context
- Displaying information about groups/tasks
- Interrupting the application
- Showing breakpoint/tracepoint status
- Getting help
- Exiting the debugger.

You can change the subset of tasks to which context sensitive commands are directed. Also, you can understand more about the current state of the application, and gain control of your application at any time, not just at user-defined breakpoints.

When a **pdbx** subset prompt is encountered, all input you type at the command line is intercepted by **pdbx**. All commands are interpreted and operated on by the home node. No data is passed to the remote nodes and standard input (STDIN) is not given to the application. Most commands in the **pdbx** subset produce information about the application and display the **pdbx** subset prompt. The exceptions are the **halt**, **back**, **on**, and **quit** commands. The **halt**, **back**, and **on** commands cause the **pdbx** prompt to be displayed when all of the tasks in the current context are in “debug ready” state.

The following example shows how the function works. A user is trying to understand the behavior of a program when tasks in the current context hang. This is a four task job with two groups defined called `low` and `high`. `Low` has tasks 0 and 1 while `high` has tasks 2 and 3. A breakpoint is set after a blocking read in task 2, and somewhere else in task 3. Group `high` is allowed to continue, and task 2 has a blocking read that will be satisfied by a write from task 0. Since task 0 is not executing, the job is effectively deadlocked and the **pdbx** prompt will not be displayed. The “effective deadlock” happens because the debugger controls some of the tasks that would otherwise be running. This could be called a debugger induced deadlock.

Using <**Ctrl-c**> allows the debugger to switch to task 0, then step past the write that satisfies the blocking read in task 2. A subsequent switch to group `high` shows task 2.

pdbx Subset Commands: The following table shows some commands that are uniquely available at the **pdbx** subset prompt, plus other **pdbx** commands that can be used. Certain commands are not allowed. The available commands keep the same command syntax as the **pdbx** subcommands (see “pdbx Subcommands” on page 1).

This subset command:	Is used to:	For more information see:
alias [alias_name string]	Set or display aliases.	“Creating, Removing, and Listing Command Aliases” on page 34
back	Return to a **pdbx** prompt.	“Returning to a pdbx Prompt” on page 19
group <action> [group_name] [task_list]	Manipulate groups. The actions are **add**, **change**, **delete**, and **list**. To indicate a range of tasks, enter the first and last task numbers, separated by a colon or dash. To indicate individual tasks, enter the numbers, separated by a space or comma.	“Grouping Tasks” on page 11
halt [all]	Interrupt all tasks in the current context that are running. If “all” is specified, all tasks, regardless of state, are interrupted. This command always returns to a **pdbx** prompt.	“Interrupting Tasks” on page 22
help [subject]	Display a list of **pdbx** commands and topics or help information about them.	“Accessing Help for pdbx Subcommands” on page 33
on <[group | task]>	Set the current context for later subcommands. This command always returns to a **pdbx** prompt.	“Setting Command Context” on page 15
source <cmd_file>	Execute subcommands stored in a file. **Note:** The file may contain context sensitive commands.	“Reading Subcommands From a Command File” on page 36
status [all]	Display the trace and stop events within the current context. If “all” is specified, all events, regardless of context, are displayed.	“Checking Event Status” on page 26
tasks [long]	Display processes (tasks) and their states.	“Displaying Tasks and their States” on page 11
quit	Exit the **pdbx** program and kill the application.	“Exiting pdbx” on page 39
unalias alias_name	Remove a previously defined alias.	“Creating, Removing, and Listing Command Aliases” on page 34

<**Ctrl-c**>	Has no effect, except to display the following message:
	`Typing Ctrl-c from the pdbx subset prompt`
	`has no effect.`
	`Use the halt command to interrupt`
	`the application.`
	`Use the quit command to quit pdbx.`
	`Type help then enter to view brief help of`
	`the commands available.`
		“Context Switch when Blocked” on page 17

Returning to a pdbx Prompt: The **back** command causes the pdbx prompt to be displayed, when all the tasks in the current context are in “debug ready” state. You can use the **back** command if you want the application to continue as it was before <**Ctrl-c**> was issued. Also, you can use it if during subset mode all of the nodes are checked into debug ready state, and you want to do normal **pdbx** processing. The **back** command is only valid in **pdbx** subset mode.

It is also possible to return to the **pdbx** prompt using the **on** and the **halt** commands.

Controlling Program Execution

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

The next few pages should act as an introduction to breakpoints and tracepoints if you are unfamiliar with **dbx**.

Refer to *IBM AIX Version 4 Commands Reference* and *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs* for more information on subcommands.

Setting Breakpoints

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

The syntax of this context sensitive subcommand is:

stop if <*condition*>

stop at <*source_line_number*> [**if** <condition>]

stop in <*procedure*> [**if** <condition>]

stop <*variable*> [**if** <condition>]

stop <*variable*> **at** <*source_line_number*>
[**if** <condition>]

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

Specifying **stop at** <*source_line_number*> causes the breakpoint to be triggered each time that source line is reached.

Specifying **stop in** <*procedure*> causes the breakpoint to be triggered each time the program counter reaches the first executable source line in the procedure (function, subroutine).

Using the <*variable*> argument to stop causes the breakpoint to be triggered when the contents of the variable changes. This form of breakpoint can be very time consuming. For better results, when possible, further qualify these breakpoints with a *source_line* or *procedure* argument.

Specify the <*condition*> argument using the syntax described by “Specifying Expressions” on page 36.

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 in a group context if the group members have different current source files. Instead, the following error message will be displayed.

   ```
   ERROR: 0029-2081 Cannot set breakpoint or tracepoint 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 24 for more information.
3. For further details on the **stop** subcommand, refer to its use on the **dbx** command as described in *IBM AIX Version 4 Commands Reference* and *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*

Interrupting Tasks

By using the **halt** command, you interrupt all tasks in the current context that are running. This allows the debugger to gain control of the application at whatever point the running tasks happen to be in the application. To a **dbx** user, this is the same as using <**Ctrl-c**>. This command works at the **pdbx** prompt and at the **pdbx** subset prompt. If you specify "all" with the **halt** command, all running tasks, regardless of context, are interrupted.

Note: At a **pdbx** prompt, the **halt** command never has any effect without "all" specified. This is because by definition, at a **pdbx** prompt, none of the tasks in the current context are in "running" state.

The **halt all** command at the **pdbx** prompt affects tasks outside of the current context. Messages at the prompt show the task numbers that are and are not interrupted, but the **pdbx** prompt returns immediately because the state of the tasks in the current context is unchanged.

When using **halt** at the **pdbx** subset prompt, the **pdbx** prompt occurs when all tasks in the current context have returned to "debug ready" state. If some of the tasks in the current context are running, a message is presented.

Setting Tracepoints

The **trace** subcommand sets tracepoints for all tasks in the current context. When any task reaches a tracepoint, it causes the debugger to print information about the state of the program for that task.

The syntax of this context sensitive subcommand is:

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

trace <*source_line_number*> [**if** <condition>]

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

trace <*variable*> [**in** <*procedure*>]
[**if** <condition>]

trace <*expression*> **at** <*source_line_number*>
[**if** <condition>]

Specifying **trace** with no arguments causes trace information to be displayed for every source line in your program.

Specifying **trace** <*source_line_number*> causes the tracepoint to be triggered each time that source line is reached.

Specifying **trace** [**in** <*procedure*>] causes the tracepoint to be triggered each time your program executes a source line within the procedure (function, subroutine).

Using the <*variable*> argument to trace causes the tracepoint to be triggered when the contents of the variable changes. This form of tracepoint can be very time consuming. For better results, when possible, further qualify these tracepoints with a *source_line_number* or *procedure* argument.

Specify the <*condition*> argument using the syntax described by "Specifying Expressions" on page 36.

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 in a group context if the group members have different current source files. Instead, the following error message will be displayed.

   ```
   ERROR: 0029-2081 Cannot set breakpoint or tracepoint 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" on page 24 for more information.

3. For further detail on the **trace** subcommand, refer to its use on the **dbx** command as described in *IBM AIX Version 4 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. 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.

```
0:@myfile.i          (from line 1 of previous example)
1:@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)
```

Deleting pdbx Events

The **delete** subcommand removes events (breakpoints and tracepoints) of the specified **pdbx** event numbers. To indicate a range of events, enter the first and last event numbers, separated by a colon or dash. To indicate individual events, enter the numbers, separated by a space or comma. You can specify “ * ”, which deletes all events that were created in the current context. You can also specify “all”, which deletes all events regardless of context. The syntax of this context sensitive subcommand is:

delete [*event_list* **|** * **|** **all**]

The event number is the one associated with the breakpoint or tracepoint. This number is displayed by the **stop** and **trace** subcommands when an event is built. Event numbers can also be displayed using the **status** subcommand. The output of the status command shows the creating context as the first token on the left before the colon.

Event numbers are unique to the context in which they were set, but not globally unique. Keep in mind that, in order to remove an event, the context must be on the appropriate task or task group, except when using the “all” keyword. 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 do one of the following:

ENTER **on** *1*

delete *0*

on *all*

delete *0*,*1*

OR

ENTER **on** *1*

delete *0*

on *all*

delete *

OR

ENTER **delete** *all*

Checking Event Status

A list of **pdbx** events can be displayed using the **status** subcommand. You can specify “all” after this command to list all events (breakpoints and tracepoints) that have been set in all groups and tasks. This is valid at the **pdbx** prompt and the **pdbx** subset prompt.

The following shows examples of **status**, **status all**, and incorrect syntax with different breakpoints set on three different groups and two tasks.

```
pdbx(all) status

all:[0] stop at "test/vtsample.c":60
```

```
pdbx(all) status all

1:[0] stop in main

2:[0] stop in mpl_ring

all:[0] stop at "test/vtsample.c":60

evenTasks:[0] stop at "test/vtsample.c":58

oddTasks:[0] stop at "test/vtsample.c":56
```

```
pdbx(all) status woops

0029-2062 The correct syntax is either 'status' or 'status all'.
```

Because the **status** command (without “all” specified) is context sensitive, it will not display status for events outside the context.

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. **Detach** is not supported in **pdbx**.

To better understand the **hook** and **unhook** subcommands, consider the following example. You are debugging a typical master/worker program containing many blocking sends and receives. You have created two task groups. One – named *workers* – contains all the worker tasks, and the other – named *master* – contains the master task. You would like to manipulate the master task and let the worker tasks process without debugger interaction. This would save you the bother of switching the command context back and forth between the two task groups.

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

ENTER **on** *workers*

- The debugger sets the command context on the task group *workers*.

ENTER **unhook**

- The debugger unhooks the tasks in the task group *workers*.

The worker tasks are still indirectly affected by the debugger since they might, for example, 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 *workers* task group, you would, assuming the context is on the *workers* task group:

ENTER **hook**

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

Note: The **hook** subcommand is actually an interrupt. When you interrupt a blocking receive, you cause the request to fail. If the program does not deal with an interrupted receive, then data loss may occur.

Examining Program Data

The following section explains the **where**, **print**, and **list** subcommands for displaying and verifying data.

Viewing Program Call Stacks

The **where** subcommand displays a list of active procedures and functions.

The syntax of this context sensitive subcommand is:

where

To view the stack trace, issue the **where** command. The following stack trace was encountered after halting task 1. You can see that the main routine at line 144 has issued an **mpi_recv()** call.

```
pdbx(1) where

read(??, ??, ??) at 0xd07b5ce0

readsocket() at 0xd07542f4

kickpipes() at 0xd0750e14

mpci_recv() at 0xd076032c

_mpi_recv() at 0xd0700e2c

MPI__Recv() at 0xd06ffab8

mpi__recv() at 0xd03c4474

main(), line 144 in "send1.f"
```

Viewing Program Variables

The **print** subcommand does either of the following:

- Prints the value of a list of expressions, specified by the *expression* parameters.
- Executes a procedure, specified by the *procedure* parameter, and prints the return value of that procedure. Parameters that are included are passed to the procedure.

The syntax of this context sensitive subcommand is:

print *expression* ...

print *procedure* ([*parameters*])

See "Specifying Expressions" on page 36 for a description of valid expressions.

Following are some examples of printing portions of a two dimensional array of *floats* in a c program which is running on two nodes.

To display the type of array `ff`, enter:

```
pdbx(all) whatis ff

  0:float ff[10][10];

  1:float ff[10][10];
```

We can see the differences in the array values across the two nodes.

To show elements 4 through 7 of rows 2 and 3, enter:

```
pdbx(all) print ff[2..3][4..7]
  0:[2][4] = 30.0000076
  0:[2][5] = 42.0
  0:[2][6] = 0.0
  0:[2][7] = -3.52516241e+30
  0:[3][4] = -3.54361545e+30
  0:[3][5] = -3.60971468e+30
  0:[3][6] = 2.68063283e-09
  0:[3][7] = 4.65661287e-10
  0:
  1:[2][4] = -1.60068157e+10
  1:[2][5] = 0.0
  1:[2][6] = 0.0
  1:[2][7] = -3.52516241e+30
  1:[3][4] = -3.54361545e+30
  1:[3][5] = -3.60971468e+30
  1:[3][6] = 2.63675126e-09
  1:[3][7] = 1.1920929e-07
  1:
```

The same results as above could be achieved by entering:

```
print ff(2..3,4..7)
```

The array `ff` is being processed within a loop with loop counters `i` and `j`. The following demonstrates printing multiple variables and using program variables to specify the array elements.

```
pdbx(all) print "i is:", i, "\tj is:", j, "\n", ff[i][j..j+1]

  1:i is: 0     j is: 1

  1: [0][1] = -3.54331806e+30

  1:[0][2] = 4.40487202e-10

  1:

  0:i is: 2     j is: 6

  0: [2][6] = 0.0

  0:[2][7] = -3.52516241e+30

  0:
```

Following are some examples which display the elements of an array of *structs*:

The command **whatis** here is used to show that the type of the variable `tree` is an array size `4` of `wood_attr_t`'s.

```
pdbx(0) whatis tree

  0:wood_attr_t tree[4];
```

Here the **whatis** command shows that `wood_attr_t` is a `typedef` for the listed structure.

```
pdbx(0) whatis wood_attr_t

  0:typedef struct {

  0:    int max_age;

  0:    int max_size;

  0:    int is_hard_wood;

  0:} wood_attr_t;
```

This **whatis** command shows that `this_tree` is a `wood_attr_t` ptr.

```
pdbx(0) whatis this_tree



  0:wood_attr_t *this_tree;
```

To display the elements of the first three entries in the tree array, enter:

```
pdbx(0) print tree[0..2]

  0:[0] = (max_age = 150, max_size = 120, is_hard_wood = 0)

  0:[1] = (max_age = 250, max_size = 150, is_hard_wood = 1)

  0:[2] = (max_age = 200, max_size = 125, is_hard_wood = 0)

  0:
```

To display the element max_size of entry 1 of the tree array, enter:

```
pdbx(0) p tree[1].max_size

  0:150
```

To display the entry that this_tree is pointing to, enter:

```
pdbx(0) p *this_tree

  0:(max_age = 200, max_size = 125, is_hard_wood = 0)
```

To display just the max_size of the entry that this_tree is pointing to, enter:

```
pdbx(0) p this_tree->max_size

  0:125
```

Following are some examples of displaying elements of a two dimensional array of *reals* in a Fortran program:

To take a look at the type of var43:

```
pdbx(all) whatis var43

 real*4  var43(4,3)
```

To display the entire array var43, enter:

```
pdbx(all) print var43

(1,1)   11.0

(2,1)   21.0

(3,1)   31.0

(4,1)   41.0

(1,2)   12.0

(2,2)   22.0

(3,2)   32.0

(4,2)   42.0

(1,3)   13.0

(2,3)   23.0

(3,3)   33.0

(4,3)   43.0
```

To display a portion of the array `var43`, enter:

```
pdbx(all)  print var43(1..2, 2..3)

(1,2) = 12.0

(2,2) = 22.0

(1,3) = 13.0

(2,3) = 23.0
```

Refer to *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs* for more information on expression handling.

Displaying Source

The **list** subcommand displays a specified number of lines of the source file. The number of lines displayed is specified in one of two ways:

Tip: Use **on** <*task*> **list**, or specify the ordered standard output option.

- By specifying a procedure using the *procedure* parameter.

  In this case, the **list** subcommand displays lines starting a few lines before the beginning of the specified procedure and until the list window is filled.

- By specifying a starting and ending source line number using the *sourceline-expression* parameter.

  The *sourceline-expression* parameter should consist of a valid line number followed by an optional + (plus sign), or – (minus sign), and an integer. In addition, a *sourceline* of $ (dollar sign) can be used to denote the current line number. A *sourceline* of @ (at sign) can be used to denote the next line number to be listed.

All lines from the first line number specified to the second line number specified, inclusive, are then displayed, provided these lines fit in the list window.

If the second source line is omitted, 10 lines are printed, beginning with the line number specified in the *sourceline* parameter.

If the **list** subcommand is used without parameters, the default number of lines is printed, beginning with the current source line. The default is 10.

To change the number of lines to list by default, set the special debug program variable, *$listwindow*, to the number of lines you want. Initially, *$listwindow* is set to 10.

The syntax of this context sensitive subcommand is:

list [*procedure* **|** *sourceline-expression*[, *sourceline-expression*]]

Other Key Features

Some other features offered by **pdbx** include the following subcommands:

- **help**
- **dhelp**
- **alias**
- **source**

Also, this section includes information about how to specify expressions for the **print**, **stop**, and **trace** commands.

Accessing Help for pdbx Subcommands

The **help** command with no arguments displays a list of **pdbx** commands and topics about which detailed information is available.

If you type “help” with one of the **help** commands or topics as the argument, information will be displayed about that subject.

The syntax of this context insensitive command is:

help [*subject*]

Accessing Help for dbx Subcommands

The **dhelp** command with no arguments displays a list of **dbx** commands about which detailed information is available.

If you type “dhelp” with an argument, information will be displayed about that command.

Note: The partition must be loaded before you can use this command, because it invokes the **dbx help** command. It is also required that a task be in “debug ready” state to process this command. After the program has finished execution, the **dhelp** command is no longer available.

The syntax of this context insensitive command is:

dhelp [**dbx_command**]

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 *workers*. 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 workers* and one for *on master*. At the **pdbx** command prompt, you would:

ENTER **alias** *mas on master*
alias *wor on workers*

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

ENTER **mas**

Likewise, you can now enter **wor** instead of **on** *workers*.

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 *wor* – created in the previous example.

`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`
`mas`	`on master`
`wor`	`on workers`
`th`	`thread`
`mu`	`mutex`
`cv`	`condition`
`attr`	`attribute`
`active`	`tasks`
`threads`	`thread`

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.

Reading Subcommands From a Command File

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

source *command_file*

The *command_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.

Notes:

1. You can also read commands from a file when starting the debugger. This is done using the **-c** flag on the **pdbx** command, or via a *.pdbxinit* file, as described in Table 2 on page 5. The *.pdbxinit* file would be a great way to automatically create your common aliases. When using a *.pdbxinit* file or the **-c** flag, you need to keep in mind that only a limited set of commands are supported until the partition is loaded.
2. STDIN cannot be included in a command file.

Specifying Expressions

Expressions are commonly used in the **print** command, and when specifying conditions for the **stop** or **trace** command.

You can specify conditions with a subset of C syntax, with some Fortran extensions. The following operators are valid:

Arithmetic Operators

+	Addition
-	Subtraction
-	Negation
*****	Multiplication
/	Floating point division
div	Integer division
mod	Modulo
exp	Exponentiation

Relational and Logical Operators

<	Less than
>	Greater than
<**=**	Less than or equal to
>**=**	Greater than or equal to
==	Equal to

=	Equal to
!=	Not equal to
< >	Not equal to
||	Logical OR
or	Logical OR
&&	Logical AND
and	Logical AND

Bitwise Operators

bitand	Bitwise AND
|	Bitwise OR
xor	Bitwise exclusive OR
˜	Bitwise complement
<<	Left shift
>>	Right shift

Data Access and Size Operators

[]	Array element
()	Array element
*****	Indirection or pointer dereferencing
&	Address of a variable
.	Member selection for structures and unions
.	Member selection for pointers to structures and unions
->	Member selection for pointers to structures and unions
sizeof	Size in bytes of a variable

Miscellaneous Operators

()	Operator grouping
(Type)Expression	Type cast
Type(Expression)	Type cast
Expression\Type	Type cast

Other Important Notes on pdbx

Initial Breakpoint

The initial automatic breakpoint, which is set by default at function main, for **pdbx** can be redefined by the environment variable **MP_DEBUG_INITIAL_STOP.** See the manual page for the **pdbx** command in Appendix A, “Parallel Environment Tools Commands” on page 193 for more information.

Overloaded Symbols

While **pdbx** recognizes function names, it is the combination of a function's name and its parameters, or the function name and the shared object it resides in, that uniquely identify it to **pdbx**. When encountering ambiguous functions, **pdbx** issues the Select menu, which lets the user choose the desired instance of the function.

The Select menu looks like this:

```
pdbx(all) stop in f1

1.ambig.f1(double)

2.ambig.f1(float)

3.ambig.f1(char)

4.ambig.f1(int)

Select one or more of [1 - 4]:
```

The **whatis** subcommand can be used to determine whether or not a function is ambiguous. If **whatis** returns more than one function definition for a given symbol, **pdbx** will consider it ambiguous.

There are a few restrictions for the **pdbx** select menu:

- All tasks in the context must have an identical view of the ambiguous function because **pdbx** will only present one menu to the user that covers all tasks. As a result, you my need to create additional groups. The view of the ambiguous function is determined by the result of the **whatis** subcommand. In the example above, *whatis f1* should have returned the same result on all tasks, in order to proceed.
- The **hook** subcommand will not restore the set of events generated by the Select menu.
- The **trace** and **print** subcommands do not support ambiguous functions within complex expressions. For example, simple expressions are always allowed:

  ```
  trace myfunc



  print myfunc(parm1, parm2)
  ```

  but complex expressions are not allowed when a function (myfunc) is ambiguous:

  ```
  trace myvar-myfunc(parm1, parm2)



  print myvar*myfunc(parm1)
  ```

Exiting pdbx

It is possible to end the debug session at any time using either the **quit** subcommand, or the **detach** subcommand if debugging in attach mode.

To end a debug session in normal mode:

ENTER **quit**

- This returns you to the shell prompt.

To end a debug session in attach mode, you can choose either **quit** or **detach**. Quitting causes the debugger and all the members of the original **poe** application partition to exit. Detaching causes only the debugger to exit and leaves all the tasks running.

ENTER **quit**

- The debugger session ends, along with the **poe** application partition tasks.

OR

ENTER **detach**

- The debugger session ends. All tasks have been detached, but stay running.

Note: You can enter the **quit** and **detach** subcommands from either the **pdbx** prompt or **pdbx** subset prompt.

Choosing **detach** causes **pdbx** to exit, and allows the program to which you had attached to continue execution if it hasn't already finished. If this program has finished execution, and is part of a series of job steps, then detaching allows the next job step to be executed.

If instead you want to exit the debugger and end the program, choose **quit** as described above.

Chapter 2. Using the pedb Debugger

This chapter describes the **pedb** debugger. The **pedb** debugger provides a simplified, Motif graphical point-and-click interface. **pedb** is designed to debug parallel C or Fortran applications. The **pedb** debugger is a **poe** application with some modifications on the *home node* to provide a user interface. This means that most of the setup for the debugger is identical to the setup for **poe**.

pedb can be used to debug an application either by starting the application under the control of the debugger, or by attaching to an already running **poe** application.

If starting the application under the control of the debugger, it is first necessary to compile the program and set the execution environment. See *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for more information on the following:

- Compiling the program. Be sure to specify the **-g** flag when compiling the program. This produces an object file with symbol table references needed for symbolic debugging. It is also advisable to not use the optimization option, **-O**. Using the debugger on optimized code may produce inconsistent and erroneous results. For more information on the **-g** and **-O** compiler options, refer to their use on other compiler commands such as **cc** and **xlf**. These compiler commands are described in *IBM AIX Version 4 Commands Reference* or your online manual pages.
- Copying files to individual nodes. Like **poe**, **pedb** requires that your application program be available to run on each node in your partition. To support source level debugging, **pedb** requires the source files to be available too, but they are only required on the home node.
- Setting up the execution environment.

If using **pedb** to attach to an application, much of the setup described above is not necessary since the application is already running. However, it is still highly desirable, but not absolutely necessary, to have the application compiled with the **-g** option. When **pedb** attaches to an application that is not compiled with **-g**, the debug information is limited to a stack trace.

As you read these steps, keep in mind that **pedb** accepts almost all the option flags that **poe** accepts, and responds to almost all of the same environment variables.

This release of **pedb** does not support the debugging of applications that were compiled with previous releases of **poe**.

Starting the pedb Debugger

You can start the **pedb** debugger in either *normal* mode or *attach* mode. In normal mode your program runs under the control of the debugger. In attach mode you attach to a program that is already running. Certain options and functions are only available in one of the two modes. Since **pedb** is a source code debugger, some files need to be compiled with the **-g** option so that the compiler provides debug symbols, source line numbers, and data type information. When the application is started using **pedb**, debugger control of the application is given to the user by default at the first executable source line within the main routine. This is function *main* in C code or the the routine defined by the *program* statement in Fortran. In Fortran, if there is no *program* statement, the program name defaults to *main*. If the file containing the main routine is not compiled with **-g** the debugger will exit. The environment variable **MP_DEBUG_INITIAL_STOP** can be set before starting the debugger to manually set an alternate file name and source line where the user initially receives debugger control of the application. Refer to the appendix on POE environment variables and command-line flags in *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*

Normal Mode

The way you start the debugger in normal mode 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, different programs can run on the nodes of your partition.

If you are debugging an SPMD program, you can enter its name on the **pedb** command line. It will be loaded on all the nodes of your partition automatically. If you are debugging an MPMD program, you will load the tasks of your partition after the debugger is started.

ENTER **pedb** [[*program*] *program options*] [*poe options*] [*X options*] [[**-I** *source directory*]...] [**-d** *nesting depth*] [**-x**]

- This starts **pedb**. You will see the **pedb** main window open. If you specified a *program*, it is loaded on each node of your partition and you see the message:

```
0030-0101 Partition loaded.
```

ENTER **pedb -a** *poe process id* [*limited poe options*] [*X options*] [[**-I** *source directory*]...] [**-d** *nesting depth*] [**-x**]

- This starts **pedb** in attach mode. See “Attach Mode” on page 43 for more information.

ENTER **pedb -h**

- This writes the **pedb** usage to STDERR. It includes **pedb** command line syntax and a description of **pedb** options.

The options you specify with the **pedb** command can be program options, POE options, or **pedb** options listed in Table 4 on page 43. Program options are those that your application program will understand. You can also specify certain X-Windows options with the **pedb** command.

For the most part, you can use the same command-line flags on the **pedb** command 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 *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*

Table 4. Debugger Option Flags (pedb)

Use this flag:	To:	For example:
-a	Attach to a running **poe** job by specifying its process id. This must be executed from the node where the **poe** job was initiated. When using the debugger in attach mode there are some debugger command line arguments that should not be used. In general, any arguments that control how the partition is set up or specify application names and arguments should not be used.	To start **pedb** in attach mode: **ENTER** **pedb -a** *<poe PID>*
-d	Set the limit for the nesting of program blocks. The default nesting depth limit is 25.	To specify a nesting depth limit: **ENTER** **pedb -d** *nesting.depth*
-h	Write the **pedb** usage to STDERR then exit. This includes **pedb** command line syntax and a description of **pedb** flags.	To write the **pedb** usage to STDERR: **ENTER** **pedb -h**
-I (upper case i)	Specify a directory to be searched for an executable's source files. This flag must be specified multiple times to set multiple paths. (Once **pedb** is running, this list can also be updated using the Update Source Path window.)	To add *directory1* to the list of directories to be searched when starting the **pedb** debugger: **ENTER** **pedb -I** *dir1* You can add as many directories as you like to the directory list in this way. For example, to add two directories: **ENTER** **pedb -I** *dir1* **-I** *dir2*
-x	Prevents stripping _ (trailing underscore) characters from symbols originating in Fortran source code. This enables distinguishing 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** **pedb -x**

Attach Mode

The **pedb** debugger provides an attach feature, which allows you to attach the debugger to a parallel application that is currently executing. This feature is typically used to debug large, long running, or apparently “hung” applications. The debugger attaches to any subset of tasks without restarting the executing parallel program.

Parallel applications run on the partition managed by **poe**. For attach mode, you must specify the appropriate process identifier (PID) of the **poe** job, so the debugger can attach to the correct application processes contained in that particular job. To get the PID of the **poe** job, enter the following command on the node where **poe** was started:

```
$ ps -ef | grep poe
```

You initiate attach mode by invoking **pedb** with the **-a** flag and the PID of the appropriate **poe** process:

```
$ pedb -a <poe PID>
```

For example, if the process id of the **poe** is 12345 then the command would be:

```
$ pedb -a 12345
```

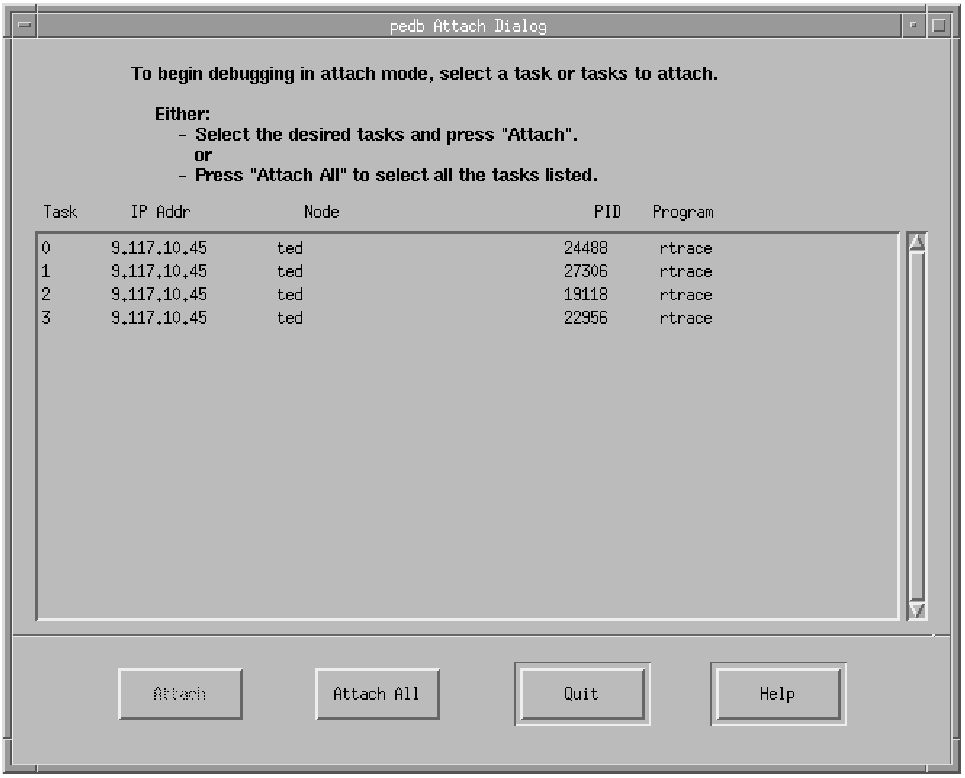
pedb starts by showing a list of task numbers that comprise the parallel job. The debugger obtains this information by reading a configuration file created by **poe** when it begins a job step. At this point you must choose a subset of that list to attach the debugger. Once you make a selection and the attach debug session starts, you cannot make additions or deletions to the set of tasks attached to. It is possible to attach a different set of tasks by detaching the debugger and attaching again, then selecting a different set of tasks.

Note: The debugger supports up to 32 nodes. When attaching to jobs larger than 32 nodes, it is suggested you select a subset of tasks less than or equal to 32.

The debugger attaches to the specified tasks. The executable is stopped wherever its program counter happens to be, and is then under the control of the debugger. The other tasks in the original **poe** application continue to run. **pedb** displays information about the attached tasks using the task numbering of the original **poe** application partition.

Attach Window

Figure 3 shows the **pedb**. Attach window.



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

Either:
- Select the desired tasks and press "Attach".
or
- Press "Attach All" to select all the tasks listed.

```
Task    IP Addr         Node                          PID    Program
0       9.117.10.45     ted                           24488   rtrace
1       9.117.10.45     ted                           27306   rtrace
2       9.117.10.45     ted                           19118   rtrace
3       9.117.10.45     ted                           22956   rtrace
```

Figure 3. pedb Attach window

The **pedb** Attach window contains a list of tasks and, for each task, the following information:

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

At the bottom of the window there are four buttons:

- Attach - causes the debugger to attach to the tasks you select. It remains grayed out until you make a selection.
- Attach All - causes the debugger to attach to all tasks listed in the window. You do not need to make any specific selections.
- Quit - closes the Attach window and exits the debugger, leaving the **poe** job running.
- Help - accesses help information about the Attach window.

At this point you can select a set of tasks to which the debugger attaches:

PRESS **Attach All** to select all tasks

OR

SELECT individual tasks by holding down the **Ctrl** key and clicking with the left mouse button.

PRESS **Attach**

- The window closes and the **pedb** Main Window appears.

Task buttons appear for each task selected for debugging. Once the debugger attaches to the selected tasks, the buttons change from a label of “UA” (unattached) to “D” (debug state), and from the default color of “wheat” to “green.”

The default group button is labelled “Attached” and consists of all the tasks chosen for attach.

When starting the debugger in attach mode, the default context is “Attached,” as indicated at the top of the main window:

```
pedb: View - 1, Context - Group Attached
```

Other Compiling Options

pedb provides substantial information when debugging an executable compiled with the **-g** option. However, you may find it useful to attach to an application not compiled with **-g**. **pedb** allows you to attach to an application not compiled with **-g**, however, the information provided is limited to a stack trace.

You can also attach **pedb** to an application compiled with both the **-g** and optimization flags. However, the optimized code can cause some confusion when debugging. For example, when stepping through code, you may notice the line marker points to different source lines than you would expect. The optimization causes this re-mapping of instructions to line numbers.

Command Line Arguments

You should not use certain command line arguments when debugging in attach mode. If you do the debugger will not start, and you will receive a message saying the debugger will not start. In general, do not use any arguments that control how the debugger partition is set up or that specify application names and arguments. You do not need information about the application, since it is already running and the debugger uses the PID of the **poe** process to attach. Other information the debugger needs to set up its own partition, such as node names and PIDs, comes from the configuration file and the set of tasks you select. See Appendix B, “Command Line Flags for Normal or Attach Mode” on page 239 for a list of command line flags showing which ones are valid in normal and in attach debugging mode.

The information in the appendix is also true for the corresponding environment variables, however **pedb** ignores the invalid setting. The debugger displays a message containing a list of the variables it ignores, and continues.

For example, if you had **MP_PROCS** set, when the debugger starts in attach mode it ignores the setting. It displays a message saying it ignored **MP_PROCS**, and continues initializing the debug session.

The pedb Main Window

As mentioned previously, you have the option of specifying the name of your application when you invoke **pedb** which causes it to be loaded on all the tasks automatically. This method is generally used to debug SPMD codes. If you need to load an MPMD code, or prefer to use the file selection window to load your partition you should not specify your application name on the **pedb** command line.

The initial **pedb** window you see will have blank areas as illustrated in Figure 4 on page 47. If you specified an application name on the command line, the debugger will continue, by loading your application for each task which will fill in the main window as illustrated in Figure 6 on page 51.

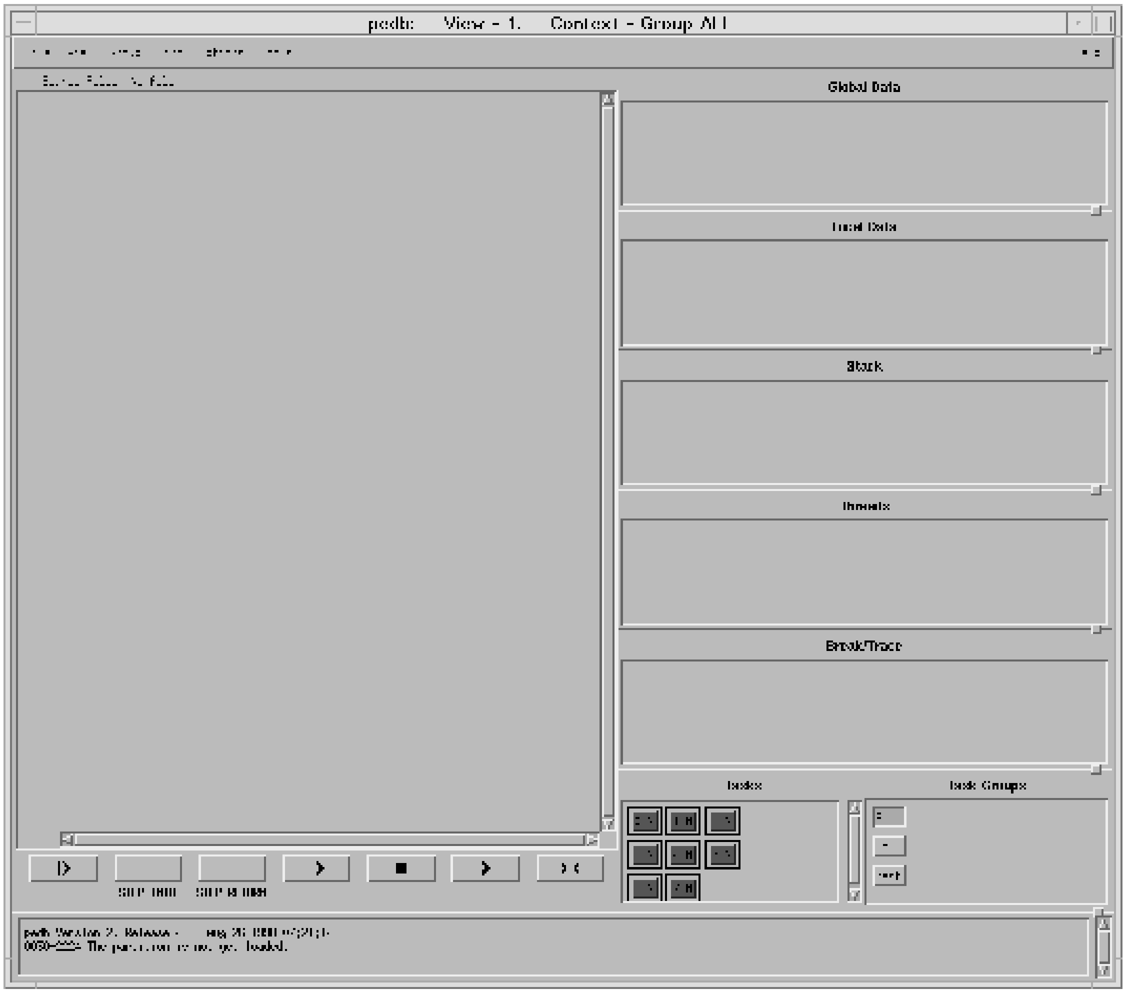
Following is a brief overview of the **pedb** main window layout.

- Across the top is a menu bar, which contains the functions you will need for debugging.

  SELECT **File** → **Load Executables ...** to choose a program to debug.

  SELECT **Find** → **Find in Source Window** to position source by search strings.

- The left half of the screen contains the source code window and the **pedb** control buttons.
  - Double click on a source line to set a breakpoint.
  - Control execution of your application with the buttons below the source code.
- Application data by task is shown in the windows on the right side of the display.
  - Global variable on request.
  - Variables local to the current block (or stack frame).
  - Calling stack listing.
  - Threads listing.
  - Event data (Break and Trace points).
- Context selection buttons at the lower right
- At the bottom, a message window.



pedb Version 2, Release [illegible] Aug 26 [illegible]
0030-[illegible] The partition is not yet loaded.

Figure 4. pedb main window before the partition is loaded

If you didn't do an SPMD load from the **pedb** command line, the initial screen opens with many options unavailable. For example, the View option and control buttons are inactive. These options will become available after all the tasks have been loaded.

During this initial loading phase, you can:

- create or delete groups
- load programs, tasks, or groups
- set a different context
- get help
- select hide/show options
- update the source path
- change context to group or task
- quit **pedb**

Loading the Partition from the Load Executables Window

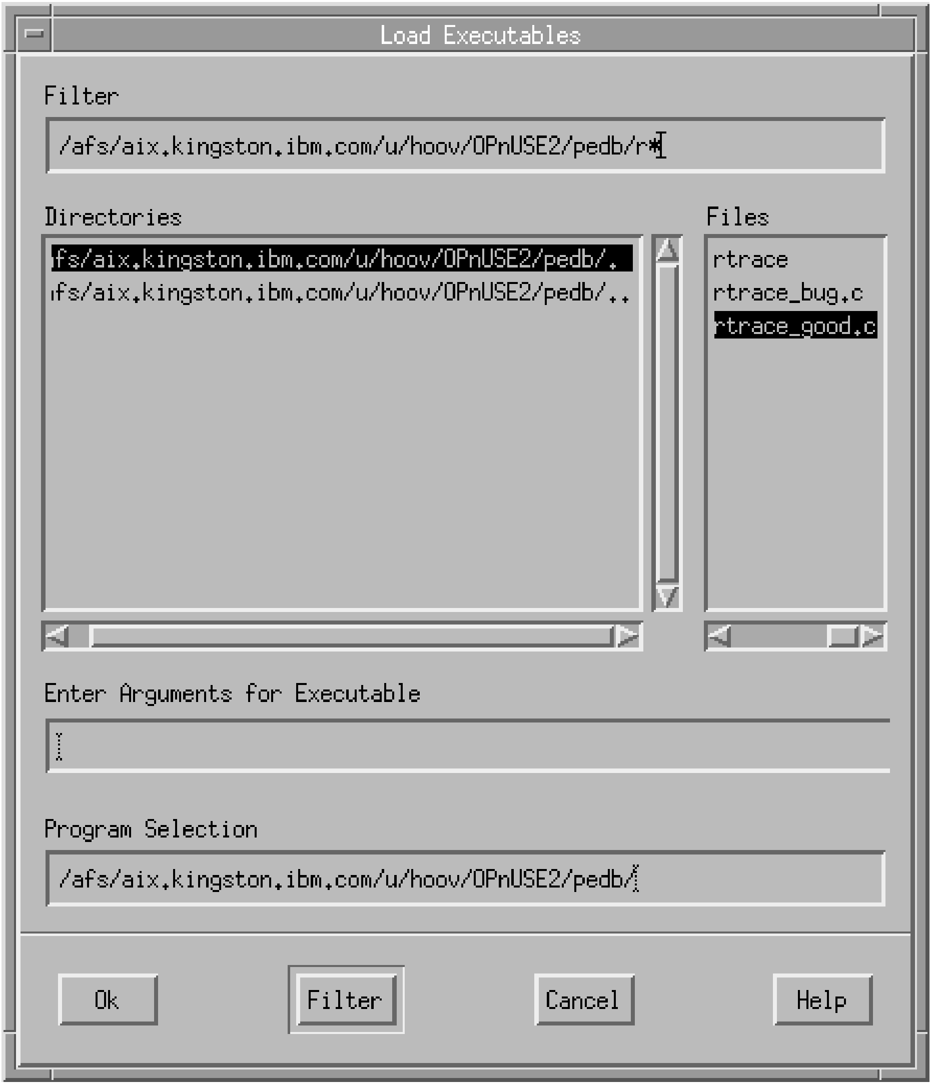
If you did not specify a program to load on the **pedb** command line, you will use the Load Executables window. In this case, a partition has been created to support the number of tasks that were defined for the application. In general, the term *task* refers to an individual program that is part of a parallel application. The number of tasks was determined by the value of the **MP_PROCS** environment variable, or the value specified by the **-procs** flag, if entered on the command line when invoking **pedb**.

A partition may be thought of as a system of one or more physical processor nodes, along with the infrastructure necessary to execute a parallel application. When you load a partition, you provide programs for the infrastructure to run.

When you specify a program to run by invoking **pedb** with a program name on the command line, it assumes an SPMD model and automatically loads all tasks with this program. With the Load Executables window however, you also have the ability to load different executables on different tasks or groups of tasks (as in the case of an MPMD application), or to load the same executable on all tasks in the instance when the file is not located in a shared file system or in the same directory on all tasks. You can load programs one task at a time by selecting a different button in the Tasks area before opening the Load Executables window. You can also load a subset of all the tasks at one time by first creating the desired task group(s), and then selecting the corresponding group button in the Task Groups area before using the Load Executables window.

Program Search Path

Like **POE**, **pedb** uses the normal shell search path that is established by the environment variable **PATH** if you don't explicitly give a path. **pedb** checks this path and loads the first occurrence of the program you specify (scanning from left to right) for each task. The mechanism for finding source files is different from this. See “Source Code Search Path” on page 101 for information on the source code search path.



Filter: /afs/aix.kingston.ibm.com/u/hoov/OPnUSE2/pedb/r*

Directories:
ıfs/aix.kingston.ibm.com/u/hoov/OPnUSE2/pedb/.
ıfs/aix.kingston.ibm.com/u/hoov/OPnUSE2/pedb/..

Files:
rtrace
rtrace_bug.c
rtrace_good.c

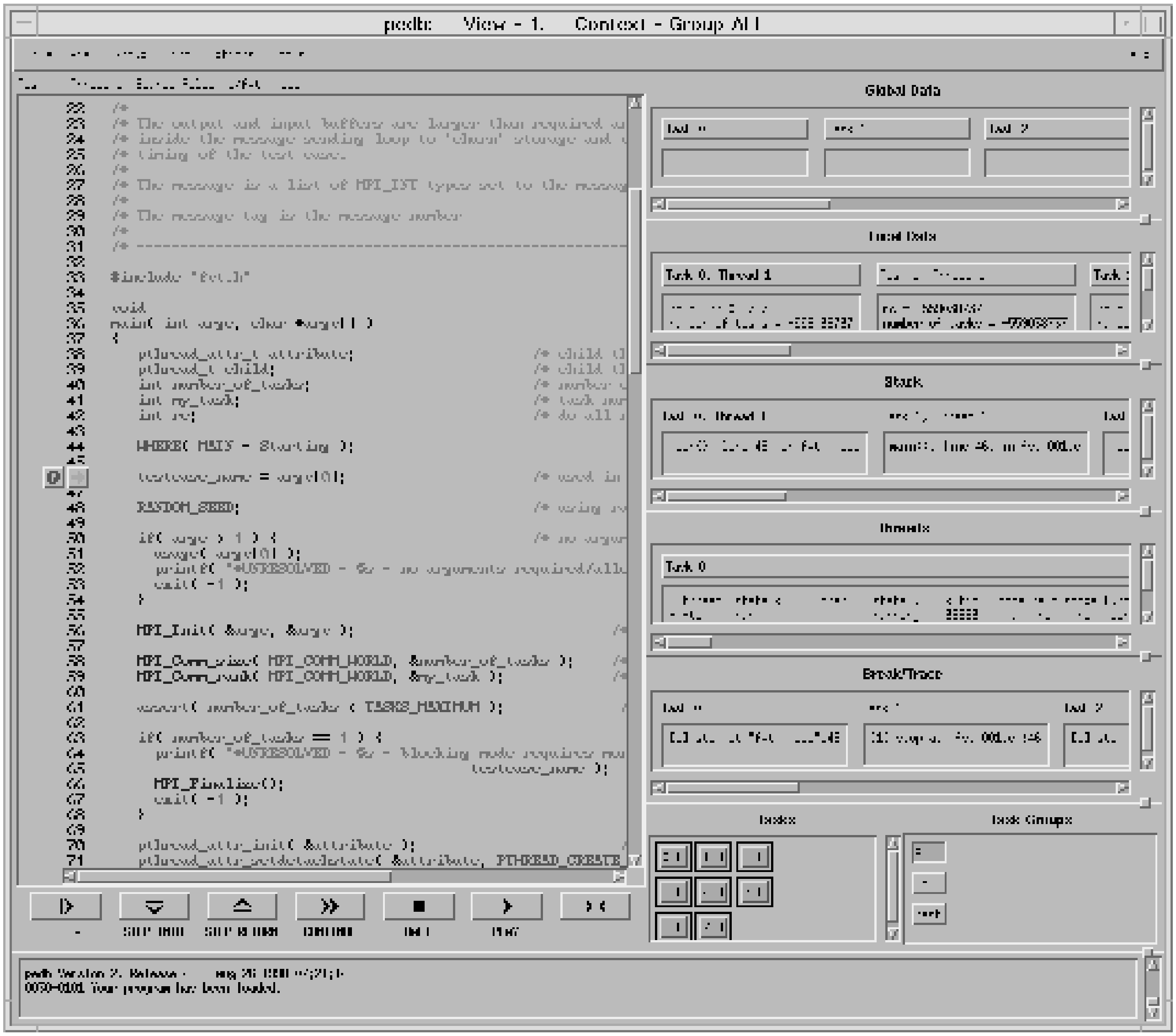
Program Selection: /afs/aix.kingston.ibm.com/u/hoov/OPnUSE2/pedb/

Figure 5. Load Executables window

To load the same executable for all tasks (SPMD):	To load different executables (MPMD):
CHECK the title bar of the **pedb** window to make sure the context is set to all tasks. This is the default when you start the **pedb** 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 Groups Area. Once the context is set on all tasks: **SELECT** **File → Load Executables ...** ● The Load Executables window opens. This window allows you to choose the appropriate directory and select the corresponding executable file. **TYPE IN** any command line arguments to the executable selected for loading. **SELECT** the directory and executable file you want to load by clicking on each name. Double clicking on the file name automatically loads the program. **PRESS** **OK** ● The Load Executables window closes, and the specified program is loaded for all tasks. Each task stops at the first executable source line. **MP_DEBUG_INITIAL_STOP** can be set to override the default of the first executable source line in *main()*. Set **MP_DEBUG_INITIAL_STOP** to the *file: linenumber*. **DISPLAYS MESSAGE** 0030-0101 Partition Loaded	Set the context before loading each program. For example, suppose there will be five tasks numbered 0 through 4. To load a program for task 0 and a program for tasks 1 through 4, you would: **PRESS** the task button labeled *0* in the Task Area. **SELECT** **File → Load Executables ...** ● The Load Executables window opens. This window allows you to choose the appropriate directory and select the corresponding executable file. **SELECT** the directory and executable file you want to load by clicking on each name. Double clicking on the file name automatically loads the program. **TYPE IN** the command line arguments to the executable selected for loading. **PRESS** **OK** ● The Load Executables window closes and the debugger loads the program for task 0. Create a group, say *group1*, containing tasks 1 through 4. **PRESS** the task group button labeled *group1* to set the context. **SELECT** **File → Load Executables ...** ● The Load Executables window opens. **SELECT** the directory and executable file you want to load by clicking on each name. Double clicking on the file name automatically loads the program. **TYPE IN** any command line arguments to the executable selected for loading. **PRESS** **OK** ● The Load Executables window closes and the debugger loads the program for tasks 1 through 4. Each tasks stops at the first executable statement. **MP_DEBUG_INITIAL_STOP** can be set to override the default of the first executable source line in *main()*. Set **MP_DEBUG_INITIAL_STOP** to the *file: linenumber*. **DISPLAYS MESSAGE** 0030-0101 Partition Loaded

The pedb Window with a Partition Loaded

Once the partition is loaded, the **pedb** window will make all of its options available.



```
/*
/* The output and input buffers are larger than required o
/* inside the message sending loop to 'clear' storage and c
/* timing of the test case.
/*
/* The message is a list of MPI_INT types set to the messag
/*
/* The message tag is the message number
/*
/* ------------------------------------------------------------

#include "fct.h"

void
main( int argc, char *argv[] )
{
   pthread_attr_t attribute;                        /* child th
   pthread_t child;                                 /* child th
   int number_of_tasks;                             /* number o
   int my_task;                                     /* task num
   int rc;                                          /* do all r

   WHERE( MAIN - Starting );

   testcase_name = argv[0];                         /* used in

   RANDOM_SEED;                                     /* using ra

   if( argc > 1 ) {                                 /* no argum
     usage( argv[0] );
     printf( "*UNRESOLVED - %s - no arguments required/allo
     exit( -1 );
   }

   MPI_Init( &argc, &argv );                                 /*

   MPI_Comm_size( MPI_COMM_WORLD, &number_of_tasks );        /*
   MPI_Comm_rank( MPI_COMM_WORLD, &my_task );                /*

   assert( number_of_tasks < TASKS_MAXIMUM );                 /

   if( number_of_tasks == 1 ) {
     printf( "*UNRESOLVED - %s - blocking mode requires mu
                                          testcase_name );
     MPI_Finalize();
     exit( -1 );
   }

   pthread_attr_init( &attribute );                           /
   pthread_attr_setdetachstate( &attribute, PTHREAD_CREATE_
```

```
pedb Version 2. Release [illegible]
0030-0101 Your program has been loaded.
```

Figure 6. pedb main window after partition is loaded

This window consists of:

- The *Title Bar*. The Title Bar is located at the top most part of the window. It identifies the view and context of the program.
- A menu bar with the following options:
  - File
  - View
  - Group
  - Find
  - Options
  - Tools
  - Help
- The *Source File Label*. This label displays the name of the source file you are currently debugging and the task number with which the source file is associated.

- 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 *Message Area*. This area displays informational and status messages about events and actions that occur. Messages about errors, warnings, and other severe conditions may *not* appear here; instead, they may appear in a message pop-up window. The contents of this message area window is controlled by a fixed-size buffer. When the buffer fills, earlier messages may no longer be accessible from the message area window. However, all error messages are duplicated in the window from which **pedb** was started.
- The *Global Data Area*. The global variable viewer displays the variables that are defined globally within the executing task(s). Global variables are only relevant when debugging C programs. For more information on global data, see “Examining Program Data” on page 68. The Global Data Area has both horizontal and vertical scroll bars for reading text displayed outside it.
- The *Local Data Area*. This area displays the values of the current routine's local variables. 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. The Stack Area has both horizontal and vertical scroll bars for reading text displayed outside it. See “Displaying Local Variables Within the Program Stack” on page 69 for more information.
- The *Threads Area*. This area displays the threads contained in the task. The Threads Area has both horizontal and vertical scroll bars for reading text displayed outside it. See “Displaying Threads Information” on page 73 for more information.
- The *Break/Trace Area*. This area displays the active Break/Trace points for the tasks in the current context. The Break/Trace Area has both horizontal and vertical scroll bars for reading text displayed outside it. See “Locating Breakpoint in Source” on page 68 for more information.
- The *pedb Execution Controls*. These controls are directly below the Source Area and allow you to control the execution of the application 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 57.
- 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, when you are defining current context. See “Setting the Context” and “Creating Task Groups” on page 54 for more information.

Setting the Context

In **pedb**, context is defined as a task or group of tasks to which the debugger directs certain actions or requests. The context sensitive controls, directly below the Source Area (lower-left), only affect those tasks in the current context. The context also determines which task's variables and stack traces will be displayed.

When you start a **pedb** session, the context is initially set to all tasks. As illustrated in Figure 4 on page 47, the title bar of the pedb window reads

pedb: View - 1, Context - Group ALL.

If you want the current context to be something other than all tasks, you can use the push buttons in the Task and Task Groups areas to change it. Press the button that corresponds to the task or group of tasks you wish to include in the current context. Note that you can select only one task or task group at a time.

For example, assume you have a parallel program that is 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 to just task 1:

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

- The **pedb** debugger sets the context to task 1. To illustrate this, the debugger highlights the task's push button and updates the title bar of the **pedb** window to read *pedb: View - 1, Context - 1*.

You can also define the current context by specifying groups of tasks. When you start a **pedb** session, a task group is automatically defined that consists of all tasks. This task group is named ALL. See “Creating Task Groups” on page 54 for information on how to create task groups.

To set the command context back to the task group ALL:

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

- The **pedb** debugger sets the context to all tasks. To illustrate this, the debugger highlights the ALL task group push button, as well as the other task push buttons, and updates the title bar of the **pedb** window to read *pedb: View - 1, Context - ALL*.

You can change the context at any time during the debugging session.

Creating and Deleting Task Groups:

In general, the term *task* refers to an individual program that is part of a parallel application.

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 Groups area. You can then set the context to include the tasks in the group by pressing its push button.

To understand why you would want to define your own task groups, consider the following example. You are debugging a master/worker 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 workers. During debugging you might start off by running the master until a blocking receive operation cannot complete. Then you could set the context on all the workers and run them past the matching send. This will allow the master task to proceed. 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 workers, you might find it helpful to group tasks 1 through 9 into a task group named *workers*. Then you would be able to press a task group button to set the context on the workers only.

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.

Provide a group name that is no longer than 32 characters which starts with an alphabetic character, and is followed by any alphanumeric character combination.

Creating Task Groups: You can create a group at any time during the debugging session using the Add Group window.

To create a task group:

SELECT **Group** → **Add Group**

- The Add Group window opens.

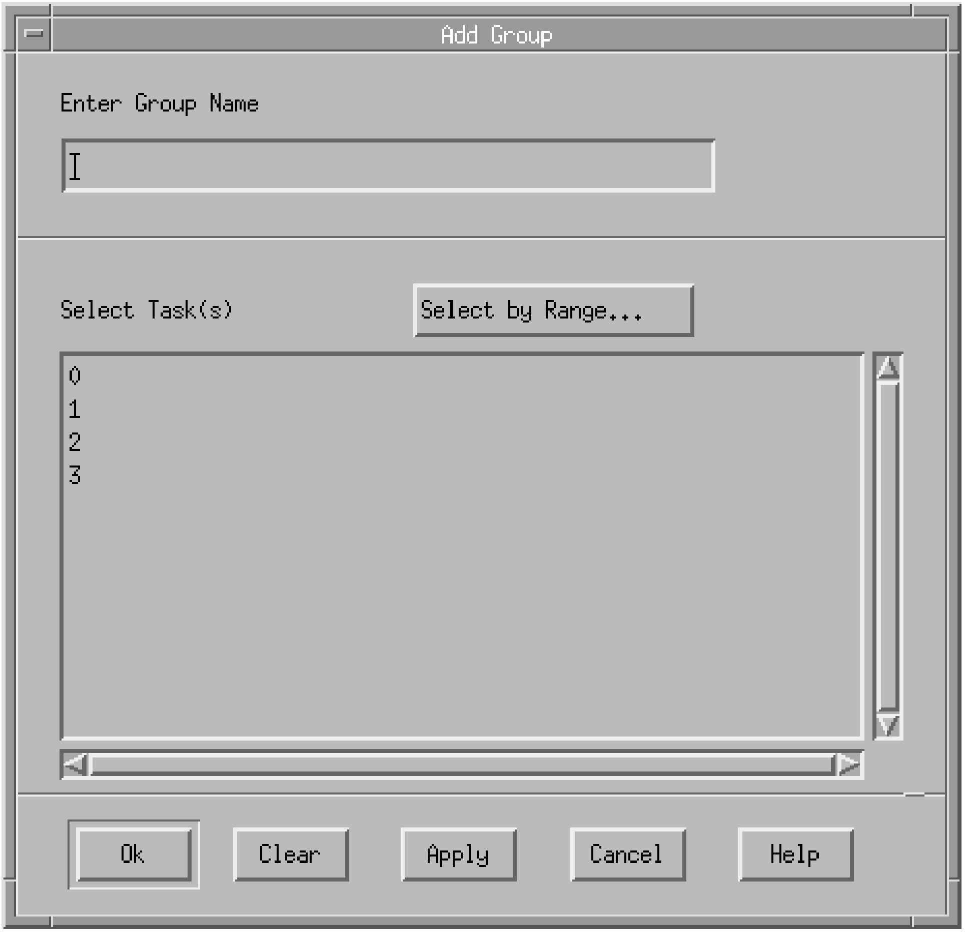


Figure 7. Add Group window

FOCUS on the Enter Group Name entry field.

TYPE IN the name of the group to be added.

FOCUS on the Select Task(s) area.

SELECT

a task by placing the cursor over a task number and depressing the left mouse button.

or

a range of tasks by depressing the left mouse button over the first task, and dragging it over the range of tasks to be included in the group.

or

a set of nonconsecutive tasks by selecting the first task, and while holding down the control key, selecting the next task(s). Note that selecting a previously selected variable will de-select it.

PRESS **Apply**

or

OK

- **Apply** creates the task group. A button containing the name of the group appears in the Task Groups area of the main window. The Add Group window remains open for further selections. **OK** creates the group, as above, and closes the Add Group window.

Clear removes all task selections and clears the text in the group name field.

Cancel closes the Add Group window.

The Select By Range feature is useful when you need to select a large range of tasks. To indicate the tasks using the Select By Range button:

SELECT the Select By Range button.

- The Select By Range window opens.

FOCUS on the **Enter range of tasks to select:** field.

TYPE IN the task list. To indicate a range of tasks, enter the first and last task numbers, separated by a colon or dash. To indicate individual tasks, enter the numbers, separated by a space or a comma.

For example:

To add:	**Type in:**
task 6 and 8	6,8
tasks 6 and 8	6 8
6 through 8, and 75	6:8 75
6 through 8, and 75	6-8 75

- **Apply** adds the selected tasks to the Add Group window and leaves the Select By Range window open for other selections.

OK adds the tasks to the Add Group window and closes the Select By Range window.

Cancel closes the Select By Range window.

Deleting Task Groups: If a particular task group no longer seems necessary, you can delete it using the Delete Group window. You may delete a group (except “all” or “attached”) at any time during the debugging session.

To delete a task group:

SELECT **Group** → **Delete Group**

- The Delete Group 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 **Apply**

or

OK

- **Apply** deletes the group and removes the group button from the Task Groups area. The Delete Group window remains open. **OK** deletes the group, as above, and closes the Delete Group window.

Cancel closes the Delete Group window.

Task Status Information: The task buttons used to change **pedb** context also display information about the status of each task. During your debug session, the color and the code letter change during different activities. For example, during the load process, the color will change from red to yellow to green and the status code letter will change from *N* to *L* to *D*.

Note: These are the default colors, but you can configure them by updating your *.Xdefaults* file.

Since each task runs independently of **pedb**, the debugger maintains status information for each task in the partition. The following table shows the status codes that are displayed in each task button and describes their meanings.

Table 5. Status Codes

Code	Default Color	Status	Description
N	Red	Not loaded	The task has not yet been loaded with an executable.
L	Yellow	Loaded	The task has been loaded with an executable.
D	Green	Debug Ready	The task is stopped and can be debugged using **pedb**.
R	White	Running	The task is in control and running.
P	MediumSeaGreen	Playing	The Play button has been depressed. The task is switching between Playing and Running, with some limited function.
x	Khaki	Exit Requested	The task in the parallel application has issued exit or returned from main, and is thus waiting in the POE specific exit code for its peer tasks to indicate that they too are waiting to exit.
X	Goldenrod	Exited	The task has reached the **hidden_exit** breakpoint that was set by **libdbx**.
E	Orange	Error	The task is in an unknown state.
U	LightSteelBlue	Unhooked	The task has been unhooked.
UA	Wheat	Unattached	The task has not yet been attached.

Controlling Program Execution

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

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

Table 6. Control Buttons

In order to:	Press this Control Button:
Manually step the execution of tasks in the current context, by a line of source code, stepping *over* subroutines and functions.	Step Over
Manually step the execution of tasks in the current context, by a line of source code, stepping *into* subroutines and functions.	Step Into
Return from the current function to the function which called it. This typically reduces the call stack by one function.	Step Return
Continue executing the tasks in the current context up to the next breakpoint or to the program's completion.	Continue
Interrupt execution of *running* tasks. The Halt button is used for situations in which a process is in a running state, such as blocked, and must be interrupted.	Halt
Have the debugger repeatedly execute the tasks. Available Play choices are Step Over, Step Into, and Continue.	Play
Break out of Play mode (as tasks finish, they will stop)	Stop
Note: To modify the function of the Play button, refer to "Customizing the Play Control Button" on page 65.	

Every time execution of a task in the current context stops, the debugger updates the **pedb** window to display the current information. In the Source Area, the debugger uses a line marker to identify the line of code at which execution has stopped. The debugger draws a line marker as an arrow 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 Fortran and C programs, this is the first executable source line defined by the user, so the debugger draws a line marker there. Since you can set the context on a subset of tasks and run them independently 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.

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 tasks and threads at that line of code. For threaded programs, each task number is followed by the displayed thread number, which is the thread whose source file, local variables, and stack are displayed. The label is visible only while you hold the mouse button down.

Line markers may be thought of as ad hoc groups which can be used to manipulate the parallel application. 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 step tasks 0 and 1 to the same line as task two. To step tasks 0 and 1 as if they were in a group:

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

PRESS the right mouse button.

- A menu appears containing four items – Step Over, Step Into, Step Return, and Continue. These menu items correspond to the Step Over, Step Into, Step Return, and Continue **pedb** control buttons.

SELECT **Step Over**

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.

The line marker, as used above, allows you to perform simple operations on single or multiple tasks that are not in a predefined task group. If tasks 0 and 1 comprised a group, you could change the context to that group, and then use the control buttons as above, then restore the original context.

Note: For more information on stepping, see “Stepping Execution” on page 63.

Threaded Programs: Each task can contain multiple threads. The threads for the tasks are listed in the threads pane on the right hand side of the **pedb** main window just below the stack pane. The list of threads for a single task can also be displayed in a separate window known as the Threads Viewer window.

When a task is in “debug ready” state, the *interrupted* thread is defined as the thread that stopped due to encountering a breakpoint or a signal. When this thread stopped, it in turn stopped all of the other threads in the process. The interrupted thread is treated specially by single step execution control. Subsequent step execution control that is issued will have the effect of stepping the interrupted thread while letting all other threads continue. It is not possible to change the interrupted thread to another thread. The interrupted thread is denoted by an asterisk at the start of the threads row.

Initially, when a task reaches “debug ready” state, the *displayed* thread is the same as the interrupted thread. The displayed thread for a task is alterable by the user. When changed to another thread, the stack, local variables, source line arrow, and the source file will be updated to reflect those for the new displayed thread. The displayed thread is denoted by a “greater than” (>) operator at the start of the threads row.

Setting Breakpoints: The **pedb** 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 using the Continue button. When the tasks reach a breakpoint, execution stops and you can then examine the state of the program.

In threaded programs, setting a breakpoint on a task, sets the breakpoint for all threads in the task. When any thread in a task hits a breakpoint, all other threads in the task are also stopped.

Note: The **Play** button will not stop execution at breakpoints, so it is suggested that you use the Continue button.

To set a breakpoint:

PLACE the mouse cursor over an executable source line in the Source Area.

PRESS the left mouse button.

- The line highlights to show that you have selected it.

PRESS the right mouse button.

- A selection menu appears.

SELECT **Breakpoint**

- The debugger sets, for each task in the current context, a breakpoint at the marked line of code.

Note: You can also set a breakpoint by double clicking the left mouse button after placing the cursor over an executable source line.

In the Source Area, the debugger places a stop marker (drawn to look like a stop sign containing an exclamation point) next to the line with the breakpoint.

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. For example:

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

Note: The debugger sets a separate breakpoint for each task in the context.

You can also specify a condition when setting a breakpoint. The task then stops executing at the breakpoint only if the condition evaluates to true.

Specifying Conditions for Breakpoints and Tracepoints: You can specify conditions with a subset of C syntax, with some Fortran extensions. The following operators are valid:

Arithmetic Operators

+	Addition
-	Subtraction
-	Negation
*****	Multiplication
/	Floating point division
div	Integer division
mod	Modulo
exp	Exponentiation

Relational and Logical Operators

<	Less than
>	Greater than
<**=**	Less than or equal to
>**=**	Greater than or equal to
==	Equal to
=	Equal to
!=	Not equal to
< >	Not equal to
||	Logical OR

or	Logical OR
&&	Logical AND
and	Logical AND

Bitwise Operators

bitand	Bitwise AND
|	Bitwise OR
xor	Bitwise exclusive OR
˜	Bitwise complement
<<	Left shift
>>	Right shift

Data Access and Size Operators

[]	Array element
()	Array element
*****	Indirection or pointer dereferencing
&	Address of a variable
.	Member selection for structures and unions
.	Member selection for pointers to structures and unions
->	Member selection for pointers to structures and unions
sizeof	Size in bytes of a variable

Miscellaneous Operators

()	Operator grouping
(Type)Expression	Type cast
Type(Expression)	Type cast
Expression\Type	Type cast

To set a conditional breakpoint:

PLACE the mouse cursor over an executable source line.

PRESS the left mouse button.

- The line 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 window.

FOCUS on the text entry field of the Conditional Breakpoint 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`. `x > 19`.

PRESS **OK**

● The debugger closes the Conditional Breakpoint 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
```

Specifying Thread Specific Breakpoints and Tracepoints: Thread specific conditions can be added to breakpoints and tracepoints using the conditional breakpoint and conditional tracepoint windows. You set the conditions using the variable **$running_thread**. This variable represents the thread that encountered the breakpoint first. This thread (the interrupted thread) will cause all of the other threads to stop. For example, adding the condition `$running_thread == 1` after selecting line `234` in your source would result in the program stopping when the thread labeled `$t1` encountered line `234`. To state this another way, the breakpoint is triggered only when thread `1` encounters the breakpoint. Refer to *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs* for more details.

Identifying the Tasks Associated with a Breakpoint: When you set a breakpoint by following the previous instructions, remember that a separate breakpoint is set for each of the tasks in the current context. If you wish to see a list 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.

When multiple breakpoints are set for a given task, the breakpoint will appear multiple times in the Break/Trace area. The corresponding breakpoint events will also be highlighted in the Break/Trace Area. This graphically shows the breakpoints that would be deleted if you used the procedure for *removing all breakpoints at the same line of code*, described in Table 7 on page 63.

Removing Breakpoints: Any number of the active tasks may have one or more breakpoints at that same line of code. You can remove:

- a breakpoint for a single task

  or
- all the breakpoints at that line of code for all tasks in the current context.

The following table shows how to remove breakpoints.

Table 7. Removing Breakpoints

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

Stepping Execution: The **pedb** 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. For threaded programs, single step execution has the effect of single stepping the interrupted thread, while letting all other non-held threads in the task continue freely without stopping at any breakpoints. All threads will again be stopped when the stepping thread reaches the appropriate source line. You can manually control each step, or have the debugger repeatedly step through the tasks in the current context at a selected interval. There are three methods of stepping the execution of your program. You can use the Step Over button to step *over* the subroutines and functions of your program. The Step Into button lets you step *into* the subroutines and functions of your program. Also, the Step Return button lets you return to the calling function.

To step execution, stepping *over* subroutines and functions:

PRESS the **Step Over** Control Button.

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

The function of the Step Over Control Button is to:

- step one line of source code
- step over functions, keeping the scope within the current function.

To step execution, stepping *into* subroutines and functions:

PRESS the **Step Into** Control Button.

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

The function of the Step Into Control Button is to:

- step one line of source code
- step into functions, following execution into called functions with debugging information.

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.

To step execution, returning to the calling function:

PRESS the **Step Return** Control Button.

● The debugger returns to the calling function and stops.

The function of the Step Return Control Button is to:

- execute the remainder of the current function
- stop in the calling function.

To automatically repeat execution:

PRESS the **Play** Control Button.

● The debugger repeatedly steps execution of the tasks in the current context.

When using the **Play** Control Button, execution continues for the tasks in the current context until you press the **Stop** Control Button. The Play function allows you to execute multiple iterations of **Step Over**, **Step Into** or **Continue** (see "Customizing the Play Control Button" on page 65 for more information). It updates the **pedb** window for each play cycle executed.

The Continue function of the **Play** Control Button can be particularly valuable when looking at loop processing. A break point may be set within a loop and the application data will be updated for each loop iteration.

Since the debugger has to keep updating the **pedb** window when in play mode, this is a slower form of execution than using the Continue Control Button. The advantage is that it provides you with more intermediate information.

For example, say you are debugging a master/worker program containing many blocking sends and receives. The context is on the worker tasks. You set a breakpoint and then press the Continue Control Button to continue execution of the worker tasks. Before reaching the breakpoint, however, the tasks hit a blocking receive intended to synchronize execution between the workers and the master task. Because the master is not in the current context, the receive operation cannot complete and so the workers cannot reach the breakpoint. Since the debugger cannot refresh the **pedb** window until the pending Continue function completes, the problem is not immediately observable. However, if you were repeatedly stepping the tasks using the Play function, you would see the line marker and other application information in effect just prior to the pending Step. You would see the task buttons holding in the *running* state and have a clear indication of where the program is hung.

To stop execution (stop playing):

PRESS the **Stop** Control Button.

● The debugger stops executing the program's tasks.

To interrupt execution (interrupt a waiting process):

PRESS the **Halt** Control Button.

● The debugger interrupts execution and returns control to the user.

Customizing the Play Control Button: When you press the Play Control Button, by default it repeatedly executes **Step Into**(s), with one-second between each **Step Into**. However, you can customize the Play Control Button to:

- select which command to repeatedly execute
- specify the delay between the command iterations in tenths of a second.

You can set these options from the main menu, using the **Options** pulldown.

To specify the command:

SELECT **Options → Change Play Command**

● Another menu appears with the command choices.

SELECT the command you want the debugger to execute repeatedly when you press the Play Control Button.

● The menu closes, and the Play Control Button is set to execute the command you specified.

To specify the delay between command iterations:

SELECT **Options → Change Play Delay**

● The Change Play Delay window opens.

FOCUS on the text entry field in this window.

TYPE IN The new delay time in tenths of seconds.

PRESS **OK**

● The Change Play Delay window closes, and the Play Control Button is set to execute its command with the new delay you specified.

You can also set these options from the pop-up menu on the Play Control Button:

To specify the command:	To specify the delay between command iterations:
PLACE the cursor over the Play Control Button. **PRESS** the right mouse button. ● The Play Menu appears. **SELECT** **Change Play Command** ● Another menu appears with the command choices. **SELECT** the command you want the debugger to execute repeatedly when you press the Play Control Button. ● The menu 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. ● The Play Menu appears. **SELECT** **Change Play Delay** ● The Change Play Delay window opens. **FOCUS** on the text entry field in this window. **TYPE IN** The new delay time in tenths of seconds. **PRESS** **OK** ● The Change Play Delay window closes, and the Play Control Button is set to execute its command with the new delay you specified.

Tracing Program Execution: The **pedb** debugger lets you set tracepoints in your program. When tasks reach a tracepoint during execution, the debugger writes information regarding the state of the program to the window from which **pedb** was invoked.

For threaded programs, tracepoints are set for all threads in the task by default. Each time a thread in the task encounters the tracepoint, a trace record is written.

Tracepoints can be set at any executable line of code within the program.

To set a tracepoint:

PLACE the mouse cursor over an executable source line.

PRESS the left mouse button.

- The line highlights to show that it is selected.

PRESS the right mouse button.

- The Break/Trace menu appears.

SELECT **Tracepoint**

- The debugger sets a tracepoint at the selected line for the tasks in the current context.

  In the Source Area, the debugger places a blue trace marker next to the line with the tracepoint. The trace marker is drawn as two eyes looking at the line of 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
```

Note: The debugger sets a separate tracepoint for each task in the context.

You can also specify a condition when setting a tracepoint. The tasks then write trace information only if the condition evaluates to true. See “Specifying Conditions for Breakpoints and Tracepoints” on page 60 for more information.

Thread specific tracepoints can be set in a similar fashion to thread specific breakpoints. See “Specifying Thread Specific Breakpoints and Tracepoints” on page 62 for more information.

To set a conditional tracepoint:

PLACE the mouse cursor over a source line.

PRESS the left mouse button.

- The line highlights to show that you have selected it.

PRESS the right mouse button.

- A selection menu appears.

SELECT **Conditional Tracepoint**

- The debugger displays the Conditional Tracepoint window.

FOCUS on the text entry field of the Conditional Tracepoint 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 Conditional Tracepoint 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] trace at "blist.f":23 if x > 19
```

Identifying the Tasks Associated with a Tracepoint: If you wish to see a list 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.

When multiple tracepoints are set for a given task, the tracepoint will appear multiple times in the Break/Trace area. The corresponding breakpoint events will also be highlighted in the Break/Trace Area. This graphically shows the tracepoints that would be deleted if you used the procedure for removing all tracepoints at the same line of code, described in Table 8.

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

  or
- all tracepoints at that line of code for all tasks in the current context.

The following table shows how to remove tracepoints.

Table 8. Removing Tracepoints

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

Unhooking Tasks: A task or group of tasks may be *unhooked* so that they execute without intervention from the debugger.

To unhook a task or group of tasks:

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

PRESS the right mouse button.

SELECT the Unhook option.

- The debugger unhooks the selected task or group of tasks. The task buttons are set to the appropriate color (the default is blue) to indicate that they have been unhooked. Note that you can change the default colors used by **pedb** by updating the X defaults file.

Examining Program Data: This section describes how to use the Data Area of the **pedb** 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.

Data, Stack, Threads, and Break/Trace Information: In the **pedb** window, the Local Data, Global Data, Stack, Threads, 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. This allows you to conserve space in the area and improve the readability of information still displayed there.

For example, say you are debugging a master/worker program. The program has 15 identical worker 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 information refreshes will be faster.

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

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

PRESS the right mouse button.

- A selection menu appears.

SELECT either **Hide Local Data**, **Hide Global Data**, **Hide Stack**, **Hide Threads**, **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 four 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.

Locating Breakpoint in Source: You can select a Break/Trace event and show the source line associated with it.

DOUBLE CLICK on an item in one of the Break/Trace window lists.

- The source window centers at the source line that is associated with the selected breakpoint and highlights that line.

OR

PLACE the mouse cursor on an item in one of the Break/Trace window lists.

PRESS the left mouse button to highlight your selection.

PRESS the right mouse button

- A menu pops up with two choices: **Delete** and **Goto Source**. Selecting **Goto Source** has the same effect as the double click described above.

Displaying Local Variables Within the Program Stack: **pedb** displays the variables that are in scope within the local program block. The Stack Area lets you display, for any of the functions or subroutines listed, the local variables that are outside the local execution block (not on the top of the stack). To display these variables:

PLACE the mouse button on a line in the Stack Area.

DOUBLE-CLICK the left mouse button.

- The line highlights to show that it is selected, and the local variables associated with the function, subroutine, or unnamed block are displayed within the Local Data. All the data variable menu options are available.

Note: Local variables for the associated tasks that are outside the local execution block (not on the top of the stack), are displayed only until you issue another Stack Area selection or execution function within **pedb**.

Displaying Local Variables and Program Stack Within a Thread: The **pedb** debugger will display program states about one thread of each task at a time. The source code, local variables, and stack trace for a task will be those of the displayed thread for the task. To select the displayed thread:

DOUBLE-CLICK on the thread.

OR

SELECT the thread from the pull-down available from each thread entry.

To select a stack entry:

PLACE the mouse button on a line in the Stack Area.

DOUBLE-CLICK the left mouse button.

- The line highlights to show that it is selected, and the local variables associated with the function, subroutine, or unnamed block are displayed within the Local Data. All the data variable menu options are available.

Understanding Data Types: In **pedb**, you can view program data through either the Global Variable Viewer or the Local Variable Viewer. These windows display a specific type of data (global or local), and the way you use them depends on the programming language.

Local Variables: The Local Variable Viewer displays the variables, in C and Fortran, that are currently visible within your local execution block.

Stepping in and out of functions and subroutines during the debugging session will alter the list of local variables that the Local Variable Viewer displays. The Local Variable Viewer displays the set of variables for the function or subroutine that is at the top of the execution stack for a particular task.

Global Variables: The Global Variable Viewer displays the variables that are defined globally within the executing task. Global variables are only relevant when debugging C programs. Unless you specifically modify it, the list of global variables displayed in the Global Variable Viewer remains constant throughout the debugging session. Initially no global variables are displayed.

Note: **xlf** and **xlc** optimize out variables that are not referenced within a program. As a result, these variables may not be available in the Global or Local Data areas.

Data Display Policies: 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 = <struct MyStruct_t>

array	with its type indicated.	a = <array 8192 of int>
		x = <array 5 struct foo>
		z = <array 10 * of int>

pointer	with its type indicated	Character Pointer examples:
		x = (nil); (unreferenced)
		x = → 0x4a567 (ptr to address)
		x = →→4; (dereferenced)
		Structure Pointer Examples:
		structx = (nil); (unreferenced)
		structx = → <struct foo> (dereferenced)

union	with its type indicated	MyOwnUnion_T = <union MyOwnUnion_T>
enum	with its value formatted according to its default type.	x = foo
logical	with its value formatted according to its default type.	do_this = .false

Viewing Variables with the Variable Viewer: Both the local and global variable windows are physically limited by the number and size of the variable information to be shown in the display. Therefore, conditions can exist where the user is unable to view all the data contained within the variable viewer due to geometry restriction on the **pedb** main window; or that the amount of data to be displayed exceeds the limitations of the window. In either of these cases, all of the variable list can be viewed using the Variable Viewer, which is an expanded form of the data window.

Initially, the variable list (local and global) is displayed in a list form, or in an iconic form. Note that when the condition of overflow occurs the variable list is replaced by the overflow icon.

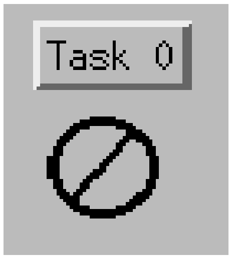


Figure 8. Overflow icon

To view the variable list in its own window,

PLACE the cursor over the task number label of the task, in the data area, of the variable you want to view.

PRESS the right mouse button.

- A pop-up menu appears with an option **Variable Viewer...**.

Note: The pop-up menu that appears for the local and global variable viewer will present a different set of options.

SELECT the **Variable Viewer** Option

- A separate window appears displaying the list of variables that was previously displayed in the local or global data area. The list (or icon) in the data areas on the main window of **pedb** should be replaced with the Variable Viewer icon.

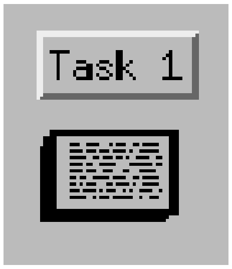
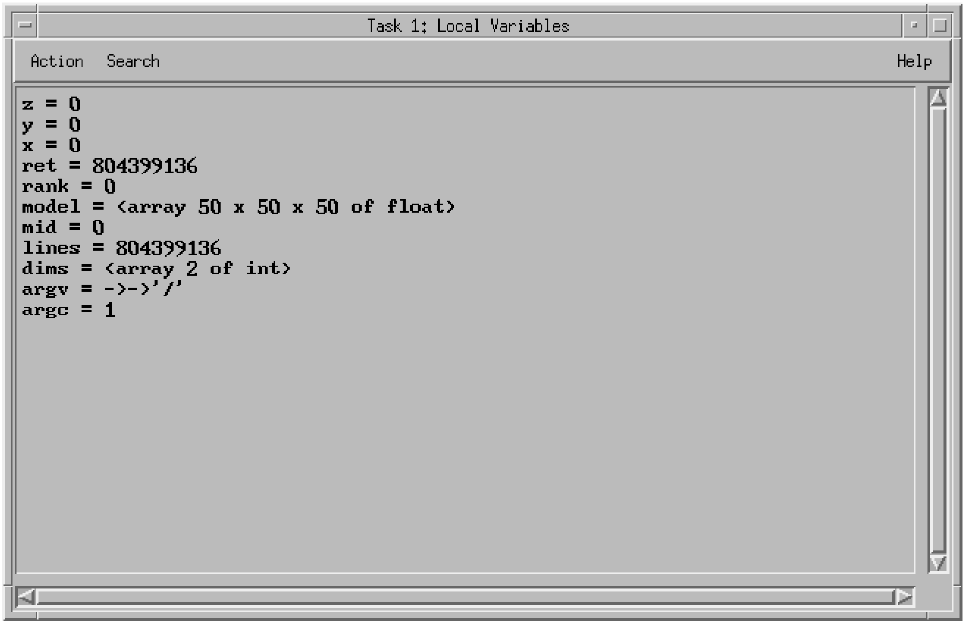


Figure 9. Variable Viewer icon

Using the Variable Viewer Window: The Variable Viewer window displays a list of global or local variables for a specified task. The task and type of data being displayed is identified by the title of the window. Figure 10 on page 72 shows the Variable Viewer window displaying local variables for the specified task. Note that all of the variable pop-up menus options that are described in “Policies for Global Variables” on page 72 are also available in this window.



```
z = 0
y = 0
x = 0
ret = 804399136
rank = 0
model = <array 50 x 50 x 50 of float>
mid = 0
lines = 804399136
dims = <array 2 of int>
argv = ->->'/'
argc = 1
```

Figure 10. Variable Viewer window

To close the variable window and return the variable list back to the main window:

PLACE the mouse cursor over the **Action** pulldown on the menubar.

PRESS the left mouse button.

● A pop-up menu should appear with a **Close** option.

SELECT the close option with the left mouse button.

● The window will close and return the contents back to the main window data area.

Note: Variables displayed in a task that is not in the current context will not be refreshed.

Policies for Local Variables: The Local Variable subwindow for each task displays all the variables within the local execution block.

Policies for Global Variables: The Global Variable subwindow requires you to explicitly select the global variables you want to view. You can do this with the Variable Selection window.

To display a global variable:

PLACE the cursor over the task number label of the task, in the Global Data area, for which you wish to choose global variables.

PRESS the right mouse button.

● a pop-up menu appears with the following options:

- Variable Selection, which provides a list from which you can select variables
- Show All, which shows all global variables for the task
- Hide All, which hides all global variables for the task
- Variable Viewer, which moves the display for the variables of this task to a separate window.

SELECT the **Variable Selection...** option.

- The Variable Selection window appears.

Note: The Variable Selection window shows only the global variables in your program that have accessible source files. If the source code of a particular program is not accessible, you may wish to use the **-I** flag, or the Source Path Window.

SELECT

a single variable by placing the cursor over it and depressing the left mouse button.

or

multiple, contiguous variables by depressing the left mouse button and dragging it over the range of variables.

or

multiple, non-contiguous variables by selecting the first variable and, while holding down the control key, selecting the next variable(s). Note that selecting a previously selected variable will de-select it.

PRESS **Apply**

or

OK

- **Apply** selects the variable(s) and leaves the Variable Selection window open. The variable(s) appear under the corresponding task label in the Global Data area.

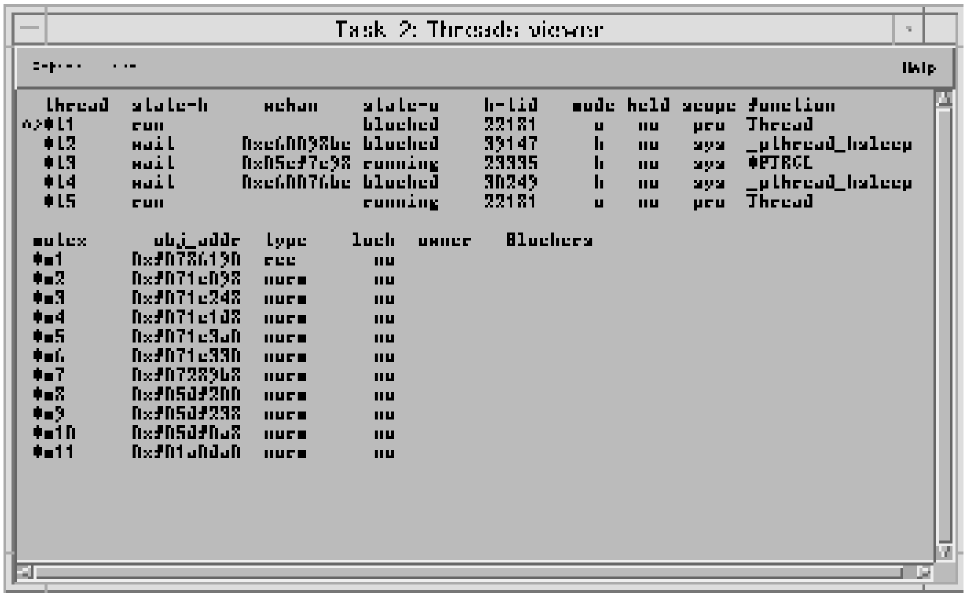
OK selects the variable(s), as above, and closes the window.

Cancel discards your selection and closes the window.

Displaying Threads Information: In **pedb**, you can display threads data for tasks in the current context. You can view a list of threads and some detailed information about the condition variables, attributes, and mutual exclusion locks pertaining to each task.

The Threads area of the **pedb** main window displays a list of threads for each task. Any one of the threads is available to select. Initially, the interrupted thread is the displayed thread. You are free to change the displayed thread for any task.

Like the local and global variable viewers, when a window representing a task in the threads area reaches a threshold, the overflow icon is displayed. See “Understanding Data Types” on page 69 for information on the local and global data viewer. At this time, you can open a Threads Viewer window, which contains the same information in the same format as in the Threads area for that task.



```
  thread   state-k     wchan     state-u    k-tid    mode held scope function
*>$t1      run                   blocked    22181     u    no   pro  Thread
  $t2      wait      0xe60098bc  blocked    39147     k    no   sys  _pthread_ksleep
  $t3      wait      0x05ef7e98  running    23335     k    no   sys  $PTRGL
  $t4      wait      0xe60076bc  blocked    30249     k    no   sys  _pthread_ksleep
  $t5      run                   running    22181     u    no   pro  Thread

 mutex       obj_addr   type    lock  owner   Blockers
 $m1       0xf0786190   rec       no
 $m2       0xf071e098   norm      no
 $m3       0xf071e248   norm      no
 $m4       0xf071e1d8   norm      no
 $m5       0xf071e3a0   norm      no
 $m6       0xf071e330   norm      no
 $m7       0xf0728968   norm      no
 $m8       0xf05df200   norm      no
 $m9       0xf05df238   norm      no
 $m10      0xf05df0a8   norm      no
 $m11      0xf01a0da0   norm      no
```

Figure 11. Threads Viewer window

The interrupted and displayed thread are denoted by "*" and ">" respectively. The ">" may move as you select different threads to display. The "*" does not change unless the program is executed and then stops again. The interrupted thread is important because it has an effect on further single stepping execution control. The displayed thread identifier is important because it denotes the thread that is represented in the stack, local variables, and source code windows.

Selecting a Thread to Display: To display a thread, go to the Threads area of the **pedb** main window:

SELECT a thread by pressing the left mouse button to highlight it.

PRESS the right button.

- The Thread menu appears.

SELECT the **Show thread** option.

- The currently highlighted thread becomes the displayed thread. The stack, local variables, and source windows will update accordingly. You can also double click on a thread with the left mouse button to display it. In this way you bypass the Thread menu.

Holding a Thread from Further Execution: You can hold a thread from execution, and then release it again for execution. To do this, enable the Thread menu as in Selecting a Thread to Display above, then:

SELECT the **Hold thread** option.

- This option controls whether the thread is dispatchable or not, and will affect subsequent execution control of the program. Held threads will not execute when single stepping or allowing the program to continue. Note that it is possible to induce hangs on other threads by holding a thread. When you select this option, the Thread menu disappears. The next time you enable the menu, the option will read "Release thread." The held field in the Threads area (of the **pedb** main window) for this thread will update appropriately when you select either of these options.

Displaying Thread Details: You can display the details of threads by opening the Threads Display menu. From the **pedb** main window:

PRESS any of the task buttons in the Threads area.

● The Threads Display menu appears.

CHOOSE **Select Display Details...**

or

Open Threads Viewer...

● The **Select Display Details...** option opens the Threads Details Selections window. From here you can choose what thread details to view. This option is available only when the task is in “debug ready” state; otherwise it is disabled. The **Open Threads Viewer...** option opens a separate window to display threads information for the task.

The Threads Details Selections Window: This window contains three toggle buttons that specify the additional thread information to be displayed in the Threads area or the Threads Viewer. You can choose any or all of:

- Display Attributes
- Display Conditions
- Display Mutexes.

After using this window to change the level of thread detail displayed, these changes you made for this task will persist across further execution control.

The Threads Viewer: The Threads Viewer is another way to view threads data. It is similar to the local and global data viewer concept. One Threads Viewer is available for each task. It exists for two reasons. First, it overcomes the limitation in the display areas on the right side of the **pedb** main window when displaying large amounts of data. Second, it provides you with a separate and larger area for displaying threads data that interests you. You can iconify the Threads Viewer window separately.

The same actions are available in the Threads Viewer as in the Thread menu. Refer to “Selecting a Thread to Display” on page 74 and “Holding a Thread from Further Execution” on page 74 for this information. There is also a find selection in the Threads Viewer window main menu bar to allow finding strings within the displayed threads data.

The Threads Viewer window menu bar has three options available: **Action**, **Find**, and **Help**.

SELECT **Action**

CHOOSE **Select Display Details...**

or

Close Viewer

● The **Select Display Details...** option opens the Threads Details Selections window, as shown in “Displaying Thread Details” on page 75. This option is only available when the task is in **debug** state, otherwise it is disabled. The **Close Viewer** option closes the Threads Viewer window, and either redisplays the contents of the window in the threads area (if there is enough room), or displays the overflow icon.

For a description of the **Find** option, see “Source File, Variable Viewer, and Threads Viewer Find” on page 102.

Threaded MPI Library: If your application uses a threaded implementation of MPI, the debugger will display the existence and status of these threads even though you may not have explicitly coded threads. Refer to *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*, and *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference* for details.

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

- Select and display a variable
- Display a variable in more or less detail
- Change a variable's value
- Change a variable's format
- Display the variable's declaration
- Select the subrange of an array.

Displaying More or Less Detail for a Variable: The **More Detail** and **Less Detail** variable options on the variable options menu operate differently depending on the data type of the selected variable. The following describes these differences.

- Simple types

  Scalers, logicals, and enumerated types have one level of detail. The name of the variable and its value are displayed by default. The **More Detail** and **Less Detail** options are not available on the Variable Options menu for these variables.

- Complex types

  Structures and unions have two levels of detail. They have their type displayed by default. To show more detail:

  PLACE the mouse cursor over the variable name, equal sign, or variable type.

  PRESS the left mouse button.

  ● The selection is highlighted.

  PRESS the right mouse button.

  ● The Variable Options menu appears.

  SELECT the **More Detail** option.

  ● This option shows the structure or union expanded with all members having their respective default levels of more or less.

  After selecting **More Detail**, the **Less Detail** option then becomes available. When selected, **Less Detail** also shows the type of the structure or union. To show less detail:

  PLACE the mouse cursor over the variable name or equal sign, and follow the same procedure as above for showing more detail.

- Arrays

  An array has three levels of detail. The first level (the default) is its type, the second displays the array elements horizontally, and the third displays the elements vertically. When displaying the second level (horizontal), rows of more than 1000 characters are broken into multiple lines.

  At the first level of detail, the **More Detail** option is available and when selected, shows the array elements horizontally. At the second level of detail, both the **More Detail** and **Less Detail** options are available. The **Less Detail** option will revert to the default of displaying the array type. The **More Detail** option, when selected, will then display the array elements vertically. At the third level of detail, the **Less Detail** option is available, and when selected, again shows the array elements horizontally.

  Note that the default is to display one element of an array. To display more array elements, the **Select Subrange** option of the Variable Options menu must be selected to bring up the Array Subrange window (see “Viewing the Contents of an Array” on page 79 for more information). This window allows selecting ranges, slices, and strides within the selected array. There is a limitation of displaying 1000 elements per array at one time.

- Pointers

  Pointers to any other type have two levels of detail. By default, the second level of detail is displayed, any dereferencing is done, and the value of the native type is displayed. To show less detail:

  PLACE the mouse cursor over anywhere from the “-” portion of the arrow and to its left.

  PRESS the left mouse button.

  ● The Variable Options menu appears.

  SELECT the **Less Detail** option.

  ● This option shows the value of the pointer in ‘hex’ format, or the string at the pointer location in the case where the native type is ‘char’.

  Pointers with multiple levels of indirection, which point to other than ‘char’ types, have a level of detail for the native type and another level of detail for each level of indirection. By default, all pointer dereferencing is done and the value of the native type is shown. To show less detail on any level of indirection:

  PLACE the mouse cursor over anywhere from the “-” portion of the arrow and to its left, and follow the same procedure as above for showing less detail.

  After any **Less Detail** option has been selected on any of the arrow pointers, subsequent selections anywhere on this variable to the right of any remaining arrows will result in bringing up a menu with the **More Detail** option available. If there are no remaining arrows, then selections anywhere on this variable will result in bringing up a menu with the **More Detail** option available. The **More Detail** option will always bring the variable back to the default of displaying the value of the native type.

Changing a Variable's Value: You can select a variable in the Data area and modify its value.

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 selection position is highlighted.

PRESS the right mouse button.

● A selection menu appears.

SELECT **Change Value**

● The debugger displays a Change Value window that corresponds to the type of variable you selected.

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

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

PRESS **OK**

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

Cancel closes the Change Value window and discards any changes.

Changing a Variable's Format: You can select a variable in the Data area and change its displayed format. You could modify the format of a variable to:

- default
- decimal
- hex
- octal
- scientific
- char
- string
- declaration

Note: Some options may be inactive, depending on the type of variable selected.

To select a variable and change its format:

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

PRESS the left mouse button.

PRESS the right mouse button.

● A selection menu appears. This menu is type-sensitive, so the only options it makes available to you are those that correspond to the type of variable you have chosen.

SELECT **Change Format**

● A selection menu appears listing the possible display formats.

SELECT the format you want.

● The debugger formats the selected variable accordingly.

Display Declaration of a Variable: After selecting a variable in the Local Variable Viewer, Global Variable Viewer, or Variable Viewer for text, you can press the right mouse button to view the Variable Selection menu. From here you can select:

Change Format → Declaration

When selected, the declaration of the variable which was previously selected is displayed after the equal sign. This is only available for scalar types.

```
Default display:

   a = 5

Display after declaration format is chosen:

   a = int a;

      or

   a = integer*4 a
```

Viewing the Contents of an Array: The Array Subrange window allows you to view the elements of an array. By defining the range of elements, you can control the portion of the array that you see. To open the Array Subrange window:

PLACE the mouse cursor over the array you want to select.

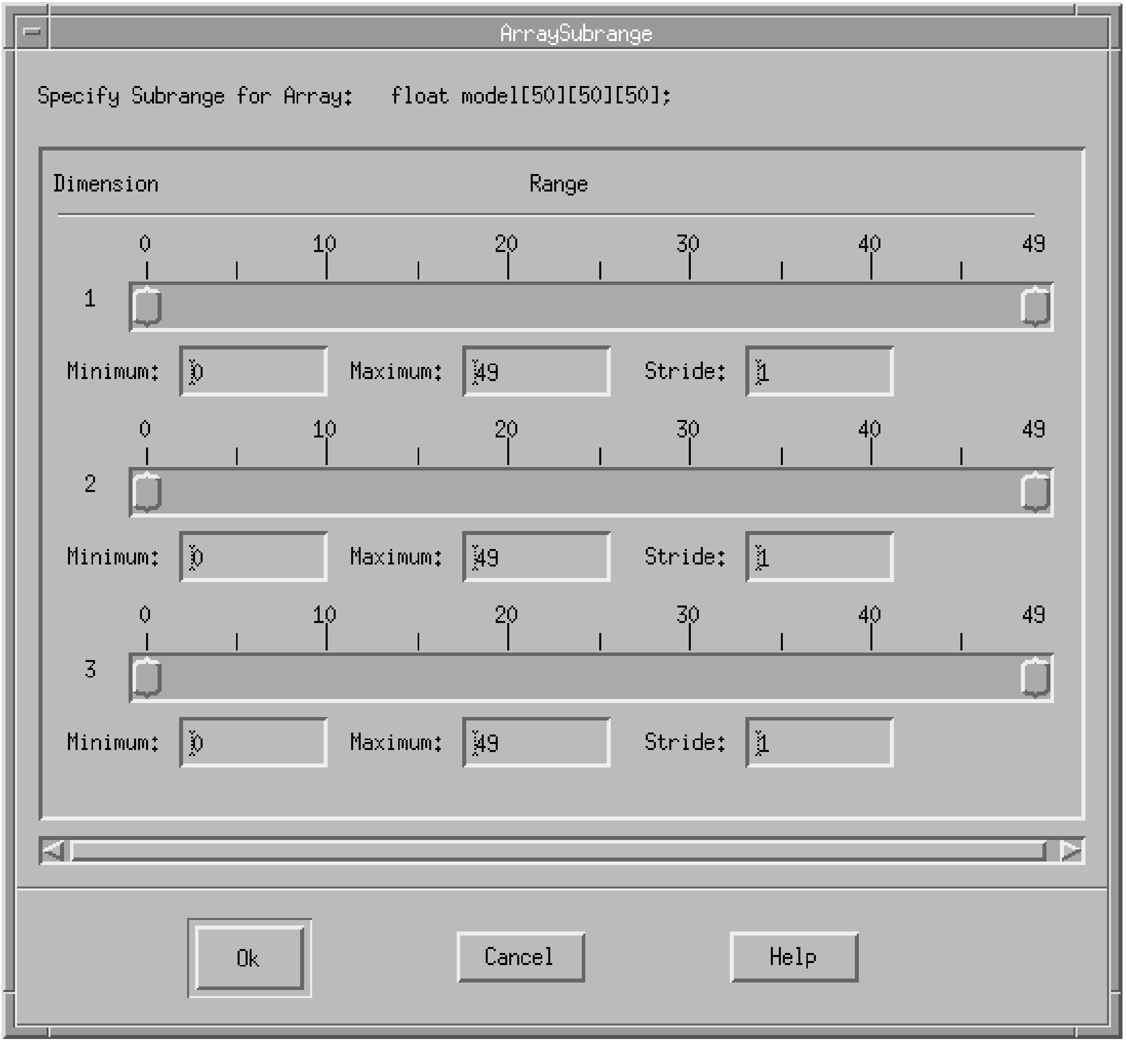
PRESS the left mouse button to highlight your selection.

PRESS the right mouse button

- the Variable Options menu appears.

SELECT **Array Subrange**

- The Array Subrange window opens, shown in Figure 12.



Specify Subrange for Array: float model[50][50][50];

1
Minimum: 0 Maximum: 49 Stride: 1

2
Minimum: 0 Maximum: 49 Stride: 1

3
Minimum: 0 Maximum: 49 Stride: 1

Figure 12. Array Subrange window

Specifying the Array Subrange: The **Specify Subrange for Array:** area of the Array Subrange window allows you to specify the set of array cells that you want to display (per dimension). Each array dimension is presented in the form of a slider, which is labelled with a scale that represents the actual range for that dimension of the array. The slider has two markers; one marks the minimum value of the subrange, the other marks the maximum value. These values are reflected in the Minimum and Maximum fields below the slider. To define a subrange for display, you choose the minimum and maximum values of the subrange individually for each dimension.

You can define a subrange in one of two ways:

PRESS the left mouse button to move the sliders horizontally left or right to mark the minimum and maximum values. These values appear in the Minimum and Maximum fields below the scale.

OR

CLICK ON the Minimum or Maximum field and clear the current value.

TYPE IN a value to define the new subrange.

The *Stride field* accepts non-zero integer values. This value allows you to skip elements for each range. The default is 1, which selects every element within the subrange for that dimension. A stride of 2 specifies every other element, and so forth. Specifying a negative stride value reverses the order of elements from which the subrange is selected. After doing this, the order goes from Maximum to Minimum, instead of Minimum to Maximum.

Note: The maximum number of array elements that can be displayed is 1000.

After you define the subrange with the appropriate values:

PRESS the **OK** button.

● The contents of the array elements that you specified are displayed, and the Array Subrange window closes. All subrange and stride specifications are retained for the next time the Array Subrange window is opened for this array on this task.

Cancel and Help Buttons

PRESS the **Cancel** button.

● The Array Subrange window closes, and all changes are discarded. All subrange and stride specifications when the window was opened are retained for the next time it is opened for this array on this task.

PRESS the **Help** button.

● Help information is displayed for the Array Subrange window.

Exporting Array Information to File: The Export window allows you to write elements of a C or Fortran array to a file in a specified data format. By defining the range of elements, you can control the portion of the array that is written to the file.

The format of the file is Hierarchical Data Format (HDF) Version 3.3. HDF is a standard format for scientific and visualization data. It was developed at the National Center for Supercomputing Applications (NCSA). See Appendix C, “Exporting Arrays to Hierarchical Data Format (HDF)” on page 241 for more information on HDF.

The Export function is only available for arrays of integer and floating point data types.

The function of the Export window is completely independent from the function of the Array Subrange window, and vice versa. This means that any specifications for a particular array in one of these windows will not affect the specifications for the same array in the other.

To open the Export window:

PLACE the mouse cursor over an array in the Local or Global variable list.

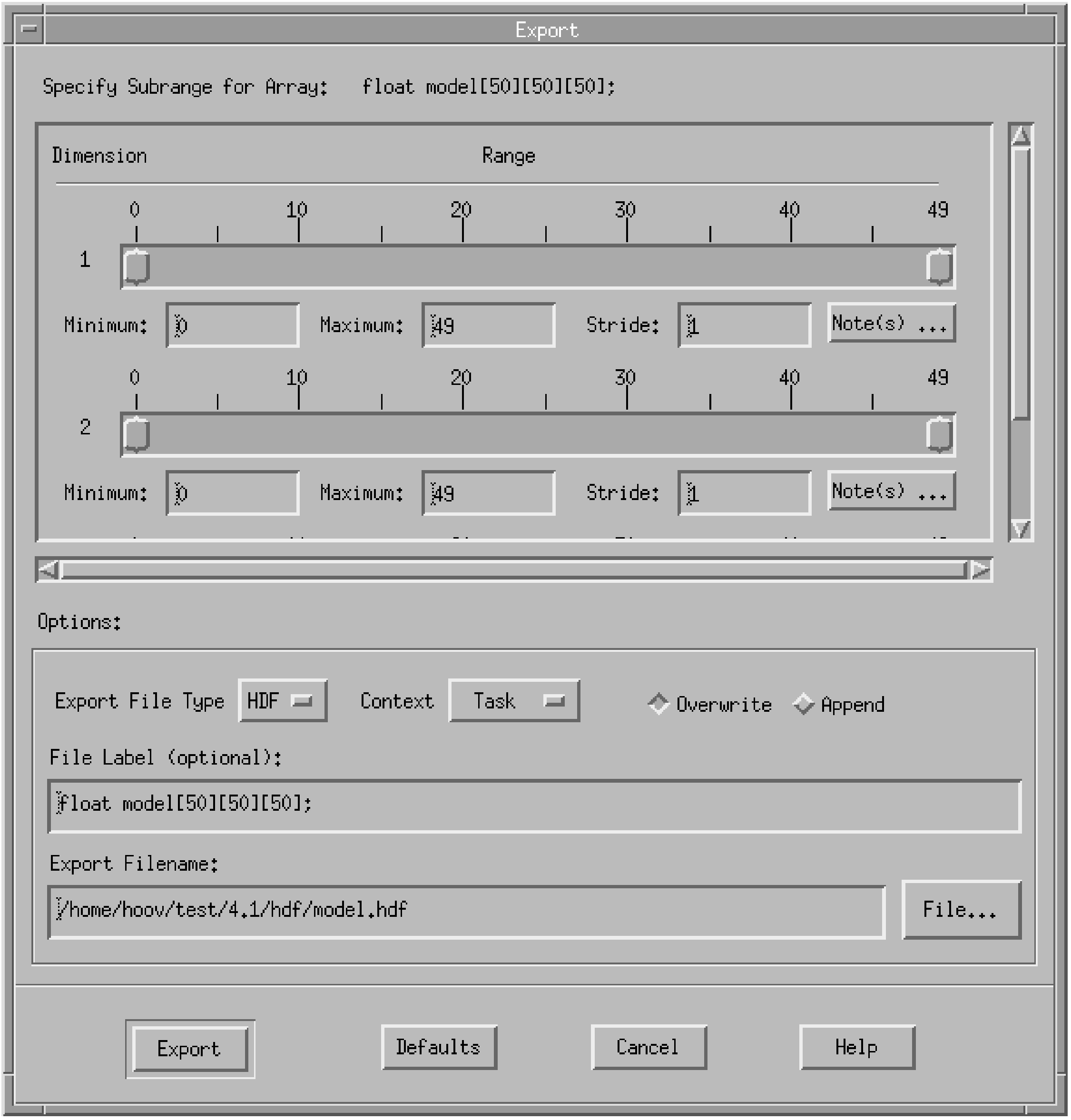
PRESS the left mouse button to highlight your selection.

PRESS the right mouse button

● The Variable Options menu appears.

SELECT **Export to File...**

● The Export window opens, shown in Figure 13.



Specify Subrange for Array: float model[50][50][50];

1 Minimum: 0 Maximum: 49 Stride: 1

2 Minimum: 0 Maximum: 49 Stride: 1

Export File Type HDF Context Task ◇ Overwrite ◇ Append

File Label (optional): float model[50][50][50];

Export Filename: /home/hoov/test/4.1/hdf/model.hdf

Figure 13. Export window

Specifying the Subrange for Export: As with the Array Subrange window, the Export window allows you to specify the subrange of an array. For details on this area, see “Specifying the Array Subrange” on page 80.

Dimension Attributes: The **Note(s) ...** pushbutton allows you to enter optional information pertaining to the associated array dimension. Clicking on this button displays the Optional Notes window, shown in Figure 14, with fields to enter attribute annotations that are specific to the output file type. To edit any of the text fields, click on the field and type in the desired text.

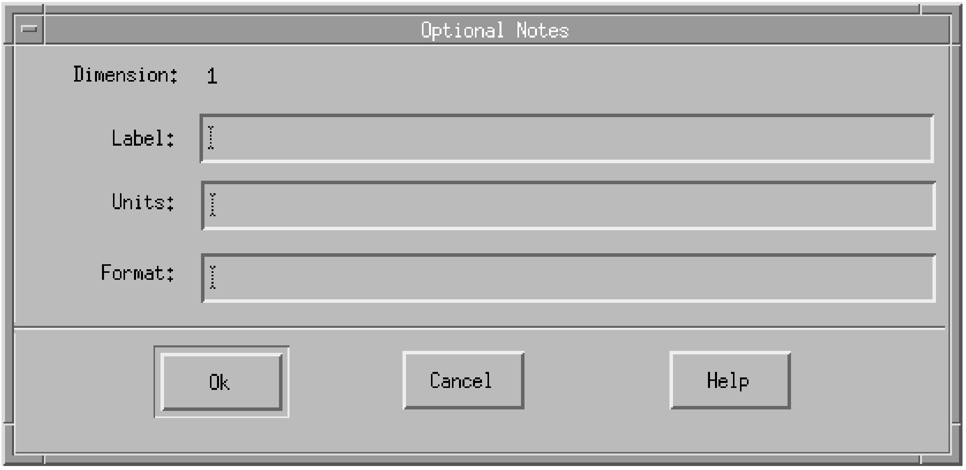


Figure 14. Optional Notes window

After you annotate the dimensions with the appropriate information:

PRESS the **OK** button.

● The notes you entered are saved and the Optional Notes window closes.

Cancel and Help Buttons

PRESS the **Cancel** button.

● The Optional Notes window closes, discarding any changes.

PRESS the **Help** button.

● Help information is displayed for the Optional Notes window.

Export Options: The following fields and buttons are available for you to specify aspects of the export:

Export File Type: Clicking on this with the left mouse button displays a menu that allows you to select the output format of the data to be written to a file. Currently, this menu has only one choice, which is HDF. This will result in the array contents being written in Hierarchical Data Format.

Context Setting: This option menu lets you select the tasks that will participate in the export, based on the criteria described below. This allows the selected array data to be written to one file in separate Scientific Data Sets, one set for each participating task.

The three choices for context are:

- "Task" - export the array data for the task from which this Export window was opened.
- "Current" - export the array data from each task within the current context. In this context the array must be displayed in all of the Global Data area task windows in the current context for all tasks to participate.
- "All" - export the array data from each of the tasks that are executing in this **pedb** debugging session. This is the same as the tasks which are included in the task group "All" if you are running **pedb** in normal mode, or the task group

“Attached” if you are running in attach mode. In this context, the array must be displayed in all of the Global Data area task windows for all tasks to participate.

If the context setting is either “Current” or “All,” the following criteria must be met for a task within the specified context to participate in the Export:

1. An array of the same name exists on each task (within the local or global block, depending on where the variable was selected).
2. The array on that task must have the same number of dimensions as the array on the task from which the Export window was opened.
3. The minimum element number for each dimension of the array must match those for the array on the task where the export is initiated. This is only a consideration with Fortran arrays, where a program can have arrays that are declared with any integer as the minimum element number. The maximum element numbers are not checked, in order to allow support for pointer variables and allocatable arrays.
4. If the array is a global array variable, then the array must be displayed in the associated task window of the Global Data area.
5. The task must be in “debug ready” state.

   If any of the tasks within the context do not meet all of the above criteria, they will be excluded from the export, and a message will be displayed to inform you of this.

   As stated above, the selected array data will be output to the HDF file as separate Scientific Data Sets. These data sets will be written to the file in order by task number.

Append and Overwrite Buttons: Clicking on the Append button results in the selected array contents to be appended to the end of an existing HDF file. This allows you to collect the contents of multiple subranges from the same array, or contents of multiple arrays, and group this information in any order within the file. Clicking on the Overwrite button causes the specified file to be created if it doesn't already exist, or be completely overwritten if it does exist. The default file writing method is Overwrite.

File Label: The File Label is a text string annotation that is written to the file just prior to writing the selected array element data for each task participating in the export. The File Label is an optional annotation which has the declaration of the array as a default text string. You can either change the label or eliminate it entirely by clicking in the text field and editing it.

Filename Specification: The **Export Filename:** field is a required field that specifies the name and path of the file that the array export data is written to. You can specify both the name of the file, and the directory where it will be located. There are two ways to do this:

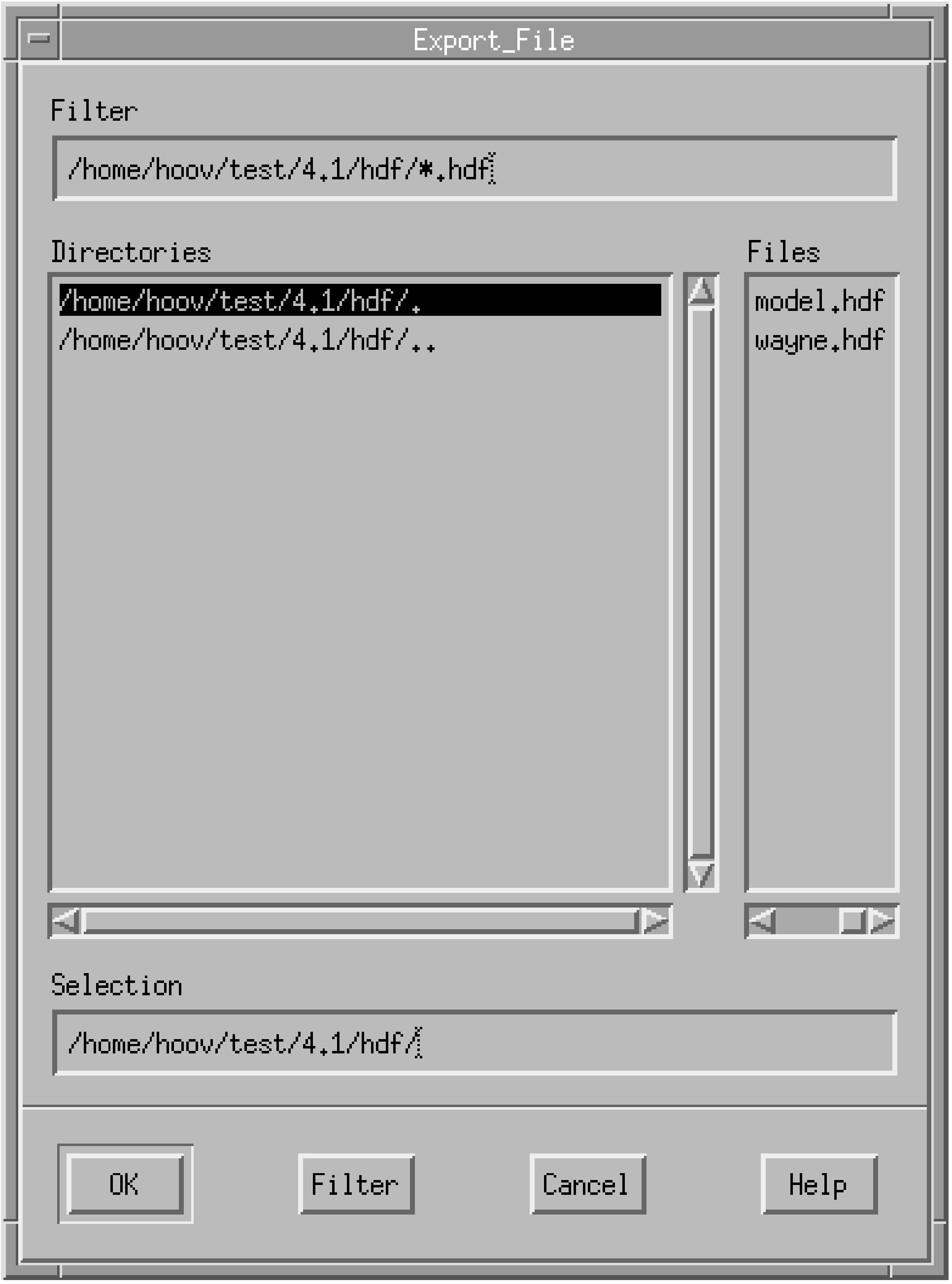
CLICK ON the **Export Filename:** field with the left mouse button, and edit the current path/file text string.

OR

CLICK ON the **File...** button located to the right of the **Export Filename:** field. This opens the Export File Selection window, that allows you to select location and file name by choosing directories and files from selection lists.

Figure 15 shows the Export File Selection window. After making these selections, the **Export Filename:** field is updated with your choice.

The **Export Filename:** field contains a default path and file name every time the Export window is opened. The default path is the directory from which **pedb** was started, and the file name consists of the name of the array that you are exporting data for, along with a .hdf suffix.



Filter
/home/hoov/test/4.1/hdf/*.hdf

Directories
/home/hoov/test/4.1/hdf/.
/home/hoov/test/4.1/hdf/..

Files
model.hdf
wayne.hdf

Selection
/home/hoov/test/4.1/hdf/

Figure 15. Export File Selection window

Export Button: Clicking on this button initiates the export of the selected array data to the specified file. The Export window remains open to allow additional exports on this array.

Stop Sign Icon: The export process can typically take more than a few seconds. When the export begins, an icon in the shape of a stop sign appears in the upper right hand corner of the Export window, and remains there until the export has completed. If you wish to stop an export that is in progress, click on this icon with the left mouse button.

Defaults Button: Clicking on this button will reset all fields and states in the Export window back to the default settings that were used the first time this window was opened for this array on this task. The subranges for all array dimensions will be set back to their full ranges.

Cancel Button: Pressing this button closes the window without exporting any data. The settings and specifications from the last export are retained for the next time the Export window is opened for this array on this task. If an export was not performed while the window was open, the settings and specifications when the window was opened for this array on this task are retained.

Help Button: This button displays help information for the Export window.

Viewing MPI Request Queues: This section describes the **pedb** debugger's *message queue* facility. Part of the **pedb** debugger interface, the message queue viewing feature is designed to help you debug Message Passing Interface (MPI) applications by showing internal message request queue information. This feature allows you to view:

- A summary of the number of active messages for each task in the application. You can select criteria for the summary information based on message type and source, destination, and tag filters.
- Message queue information for a specific task.
- Detailed information concerning a specific message.

Initial Error and Warning Messages: It is possible that there could be problems when **pedb** does it's initial checking. For example:

- The version of MPI being used may not be supported by the version of the debugger.
- The application may be using the non-threaded version of MPI, which the debugger does not support.

If either of these problems are discovered, an error message will appear, and the message queue debugging window will not appear, or will close.

When you start the message queue facility, it is possible that MPI has not initialized yet. If this is true, the initial message queue window will indicate that there is no data. The following describes this case in more detail.

During the initial checking, the debugger also determines the mode in which the MPI application is running. If it is not running in “debug” mode, the data will not include information on blocking messages. Debug mode is achieved by setting the **MP_EUIDEVELOP**environment variable to *DEB*. For more information on **MP_EUIDEVELOP**, refer to *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*. If the application is not running in debug mode, a warning message will be displayed along with the initial message queue debugging window.

Requesting information for any of the message queue windows causes the cursor to change to a “stopwatch” Further requests are disabled until the current request has finished. While the stopwatch is showing, the **pedb** main window is disabled. If a problem occurs, or a request is taking too much time, the message queue windows can be cancelled by pressing the **Cancel** button on the Application Message Queues window. This closes all message queue windows and enables the **pedb** main window. Pressing the **Cancel** button on the Application Message Queues window will always have the effect of closing all the message queue debugging windows.

Note: You can customize the colors used in any of the message queue windows. You can define these resources in the **pedb** X defaults file **/usr/lpp/ppe.pedb/defaults/Pedb.ad**.

Application Message Queues Window: To start the message queue facility, from the **pedb**main menu area:

SELECT **Tools → Message Queues**

- The Application Message Queues window opens.

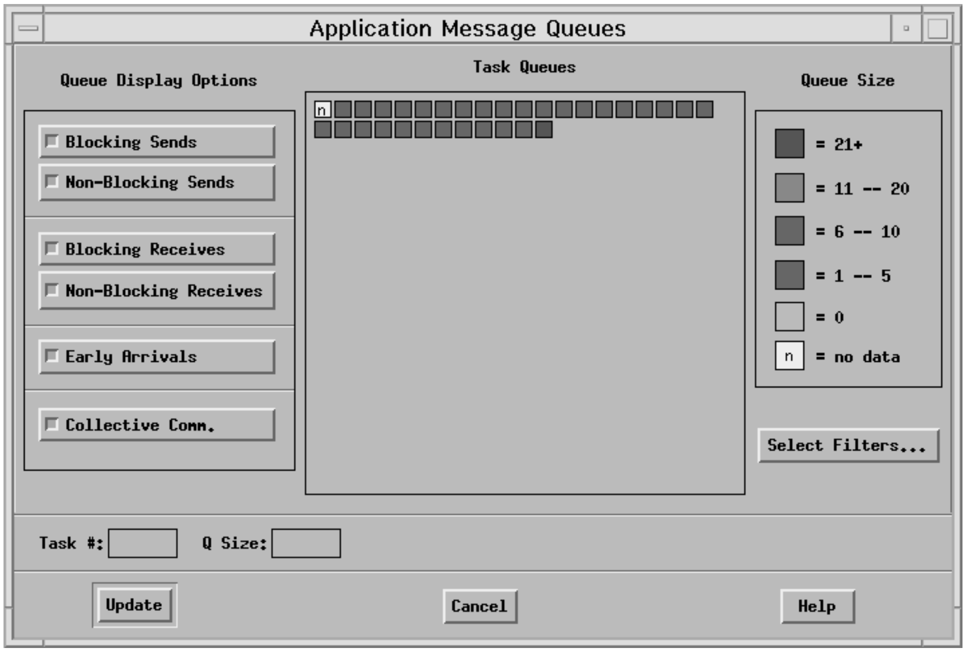


Figure 16. Application Message Queues window)

The Application Message Queues window displays message queue summary information and provides a starting point for additional message queue debugging features.

The center of the Application Message Queues window contains a set of buttons representing all the tasks defined for the MPI application. The color of each button indicates the approximate size of the queue. The interpretation of the colors is given in the Queue Size scale on the right side of the window.

To further assist in interpreting these buttons, there is a message area at the bottom of the window containing the task identifier and the actual queue size. These values are filled in automatically when you move the cursor over one of the task buttons. On the right side of this area, informational messages appear at appropriate times.

This window also contains a list of Queue Display Options on the left side, and a Select Filters... button on the right, under the Queue Size scale. The criteria for selecting summary information can be modified by selecting these options and filters. Once you set option and filter information, it remains set for future updates until you change it.

Note: You should use caution when leaving criteria set, since this can incur considerable overhead, especially in the case of filters.

See “Selecting Options” on page 88, and “Select Filters Window” on page 88 for more information.

When you first open this window, it contains summary information that includes all types of messages, including *early arrivals*. Early arrivals refer to messages that have arrived at a task, but have no posted receives to accept them.

A button with a tan colored (default color) background and labeled with an “n” indicates there is currently no data available for that task. There could be many reasons for no available data. Basically, there is no data unless a task is in “debug” state, as indicated by the task buttons at the bottom of the **pedb** main window. Also, no data is available unless a task is in the current debugger context. The reason for a task not having any data is displayed in the right side of the message area when you move the cursor over the task button.

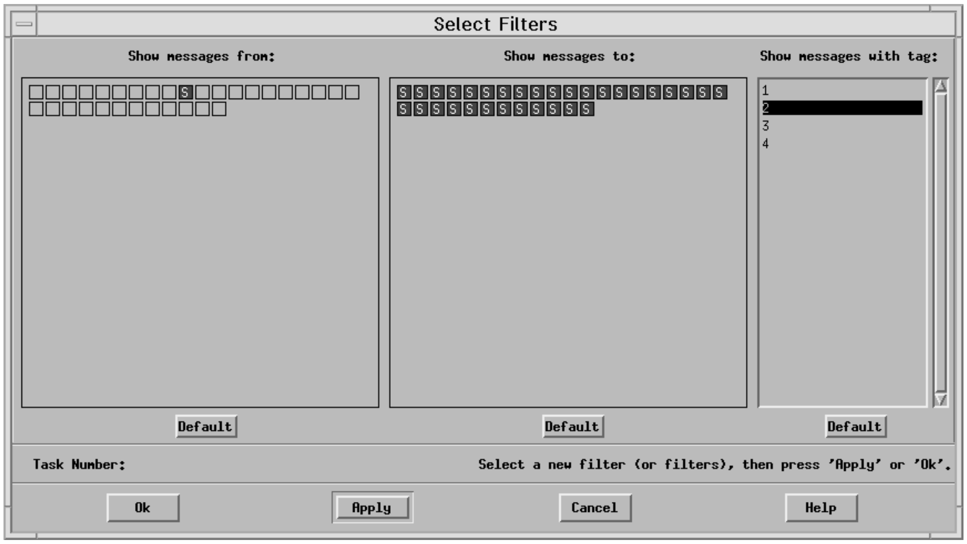
Pressing the Cancel button on this window causes all message queue windows to close.

Selecting Options: On the left side of the window there is a series of toggle buttons representing different categories of messages. When you first open the window, all the toggles are selected indicating the summary information is being collected for all categories. You can select or deselect any combination of these options for future summary information. Once you make the selections, you can retrieve the summary information by pressing the Update button at the bottom of the window.

Select Filters Window: Another way to set the criteria for gathering summary information is by using the Select Filters window. To open this window:

PRESS **Select Filters...**

- The Select Filters window opens.



Select a new filter (or filters), then press 'Apply' or 'Ok'.

Figure 17. Select Filters window

The Select Filters window is used to set source, destination, and tag values as selection criteria for message queue summary information.

This window consists of three categories of filters you can set:

- Show messages from: (source area)
- Show messages to: (destination area)
- Show messages with tag:

Both the source and destination areas contain a set of buttons representing tasks defined for the application. The corresponding task numbers are displayed in the bottom left hand side of the window when you move the cursor over the buttons. You can select one task by pressing the task button. You can select multiple contiguous tasks by pressing and holding the left mouse button and moving the pointer across the desired tasks. Also, you can select multiple non-contiguous tasks by holding down the **Ctrl**key and selecting individual or contiguous tasks.

The “Show messages with tag:” list on the right side of the window is created by extracting all the unique tags from the message records of the available tasks. The way you select tags is similar to the way you select the other filters, except lines of the tag list are selected instead of task buttons.

You can select one or more values from each category. If you do not want a specific setting, press the default button to select all possible values. Any value found in a message is acceptable and meets the criteria. You should select at least one value for each category. In other words, a message record satisfies the filter criteria if it has one source value, one destination value, and one tag value. The window opens displaying the current settings.

Note: Once the filters are set, you need to go back to the Application Message Queues window to request an update.

Scale Range Setting Window: The default queue size ranges on the Application Message Queues window may not be appropriate for all MPI applications. You can adjust the top three ranges to more reasonable sizes. To modify the range for a particular button:

SELECT the button with the mouse to open a Scale Range Setting window.

The color of the small square in this window corresponds to the color of the range button, to help you keep track of which button is being changed. To change the minimum value:

ENTER the new value.

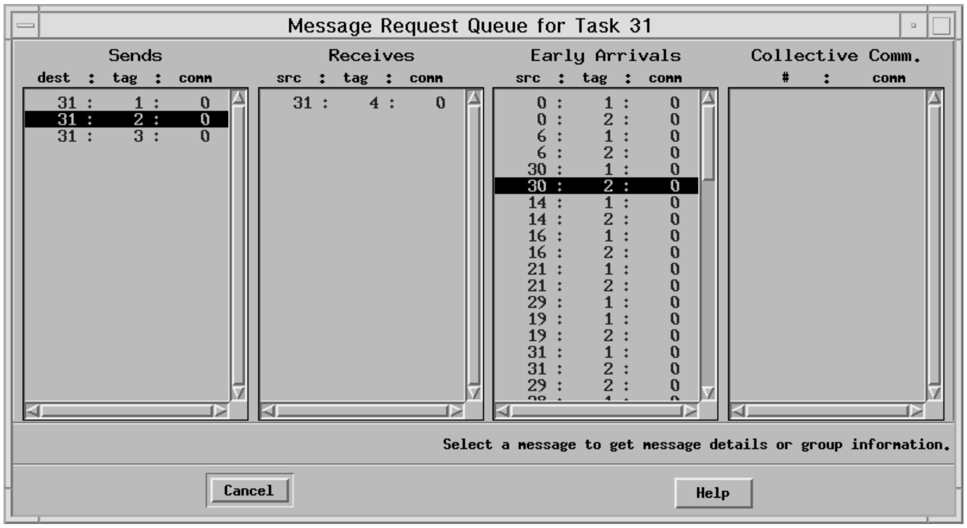
PRESS **OK**

● The minimum value of this range and the maximum value of the previous lower range is adjusted. If you attempt to set the minimum value lower than the previous range minimum value plus one, you will get an error message. You are responsible for not defining overlapping maximum ranges.

Task Message Queue Window: You can also get further information on a particular task's message queue by opening the Task Message Queue window. On the Application Message Queues window:

SELECT the task button for the desired task with the mouse.

● The Task Message Queue window opens. The number of the task the cursor is on and the actual queue size is displayed in the lower left side of the Application Message Queues window.



```
Sends                Receives             Early Arrivals       Collective Comm.
dest : tag : comm    src : tag : comm     src : tag : comm       #  :   comm
 31 :   1 :   0      31 :   4 :   0         0 :   1 :   0
 31 :   2 :   0                             0 :   2 :   0
 31 :   3 :   0                             6 :   1 :   0
                                            6 :   2 :   0
                                           30 :   1 :   0
                                           30 :   2 :   0
                                           14 :   1 :   0
                                           14 :   2 :   0
                                           16 :   1 :   0
                                           16 :   2 :   0
                                           21 :   1 :   0
                                           21 :   2 :   0
                                           29 :   1 :   0
                                           19 :   1 :   0
                                           19 :   2 :   0
                                           31 :   1 :   0
                                           31 :   2 :   0
                                           29 :   2 :   0
```

Select a message to get message details or group information.

Figure 18. Task Message Queue window

The Task Message Queue window gives a list of the message request records and early arrival records for the task. This window displays the queue of currently active messages for this task. The information is separated into four categories:

- Sends
- Receives
- Early Arrivals
- Collective Communications.

Each entry represents a unique message. Entries in the first three categories provide the tag, communicator, and source or destination of the message. The source and destination is given in terms of task id. Entries in the Collective

Communications column contain an arbitrary message index and the communicator. Blocking message entries are printed in red and non-blocking messages are printed in blue.

The early arrival messages are unique in that they represent messages sent by a task that have arrived before a receive has been posted to accept them.

From this window you can get additional message details or message communicator and group data. To view this data:

SELECT a particular entry with the left mouse button.

HOLD DOWN the right mouse button.

- The Message Data menu opens.

SELECT **Message Details**

or

Group Info

Message Details Window: To open the Message Details window:

SELECT **Tools** → **Message Queues** from the **pedb** main window to get the Application Message Queues window.

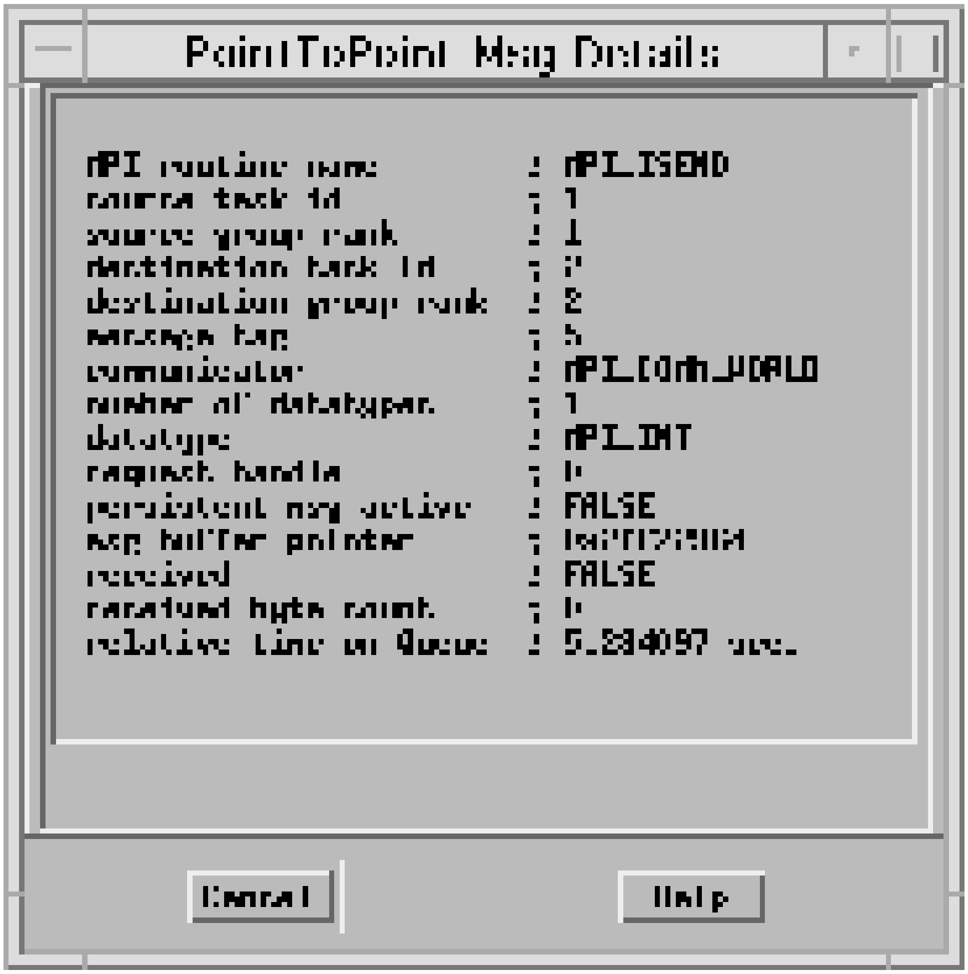
SELECT the desired task button to get the Task Message Queue window.

SELECT the desired message entry.

HOLD the right mouse button to get the Message Data menu.

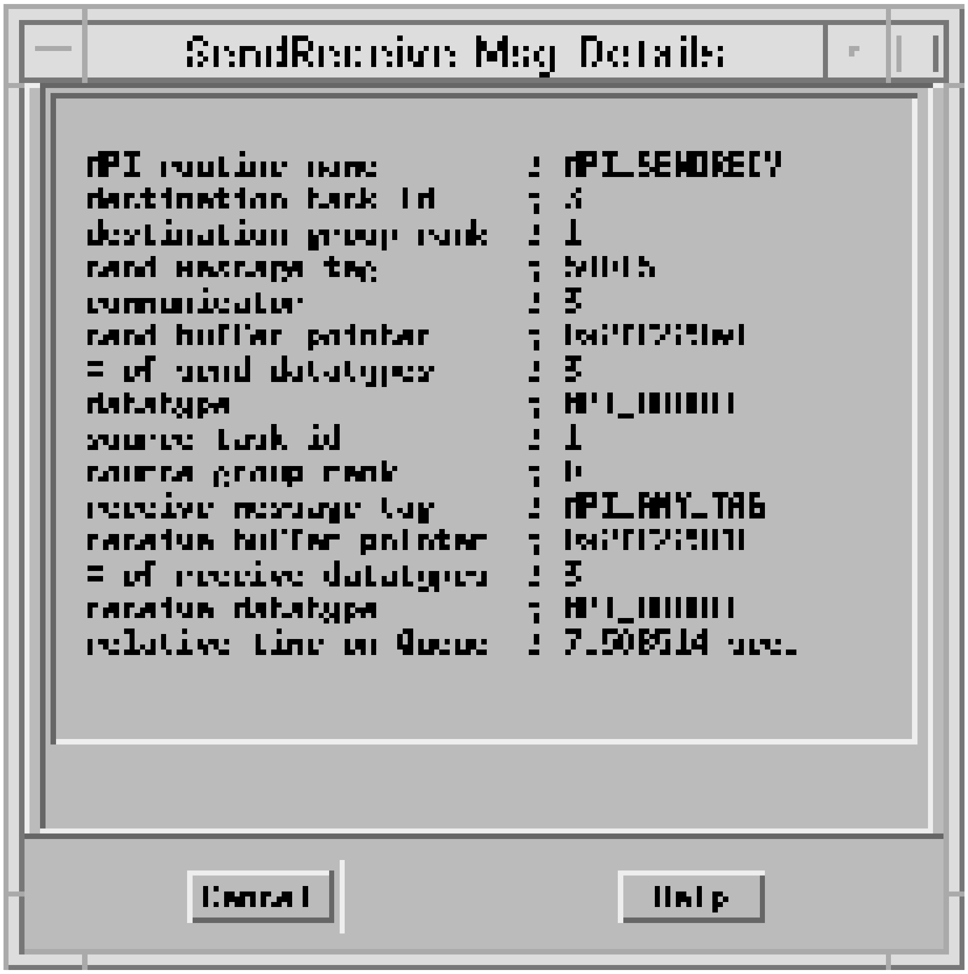
SELECT **Message Details** from the Message Data menu.

This window displays details for a specific message. There are four basic formats to this window corresponding to point to point, send/receive, collective communication, and early arrival messages. The following figures show examples of each window.



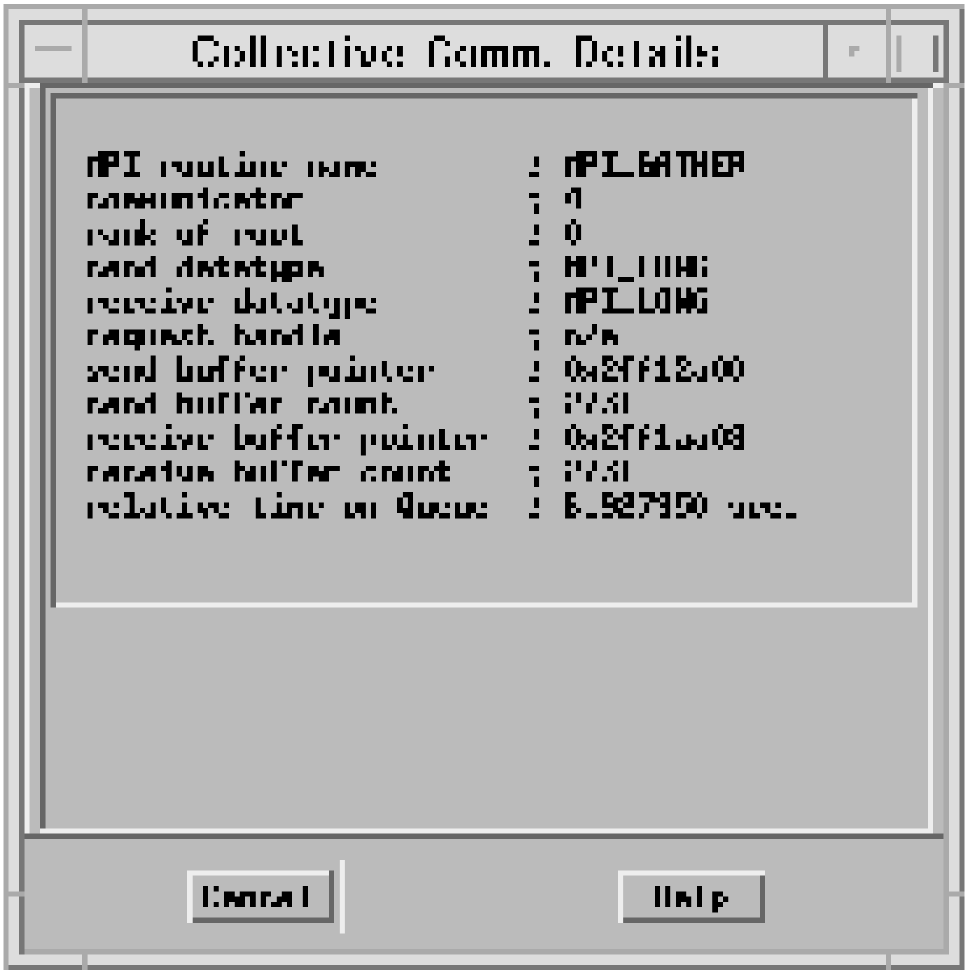
```
MPI routine name          : MPI_ISEND
source task id            : 1
source group rank         : 1
destination task id       : 2
destination group rank    : 2
message tag               : 5
communicator              : MPI_COMM_WORLD
number of datatypes       : 1
datatype                  : MPI_INT
request handle            : 0
persistent msg active     : FALSE
msg buffer pointer        : 0x2ff22504
received                  : FALSE
received byte count       : 0
relative time on Queue    : 5.284097 sec.
```

| *Figure 19. Point to Point Message Details window*



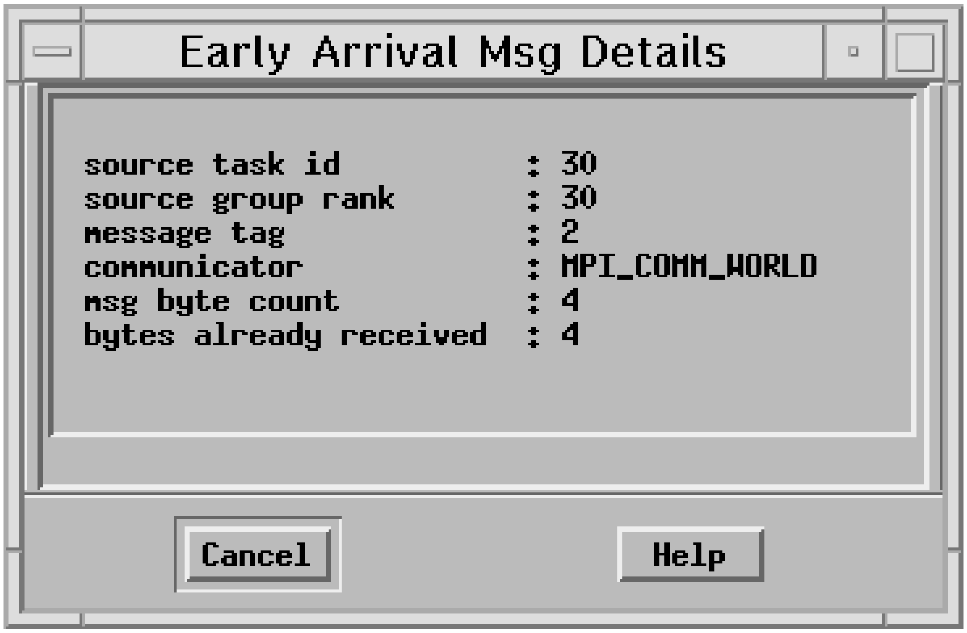
```
MPI routine name         : MPI_SENDRECV
destination task id      : 4
destination group rank   : 1
send message tag         : [illegible]
communicator             : 3
send buffer pointer      : [illegible]
# of send datatypes      : 3
datatype                 : MPI_INTEGER
source task id           : 1
source group rank        : 0
receive message tag      : MPI_ANY_TAG
receive buffer pointer   : [illegible]
# of receive datatypes   : 3
receive datatype         : MPI_INTEGER
relative time on Queue   : 7.508514 sec.
```

| *Figure 20. Send/Receive Message Details window*



```
MPI routine name         : MPI_GATHER
communicator             : 4
rank of root             : 0
send datatype            : MPI_LONG
receive datatype         : MPI_LONG
request handle           : n/a
send buffer pointer      : 0x2ff18a00
send buffer count        : 2731
receive buffer pointer   : 0x2ff1aa08
receive buffer count     : 2731
relative time on Queue   : 6.527950 sec.
```

Figure 21. Collective Communications Details window



```
source task id           : 30
source group rank        : 30
message tag              : 2
communicator             : MPI_COMM_WORLD
msg byte count           : 4
bytes already received   : 4
```

Figure 22. Early Arrival Message Details window

Message Group Information Window: To open the Message Group Information window:

SELECT **Tools** → **Message Queues** from the **pedb** main window to get the Application Message Queues window.

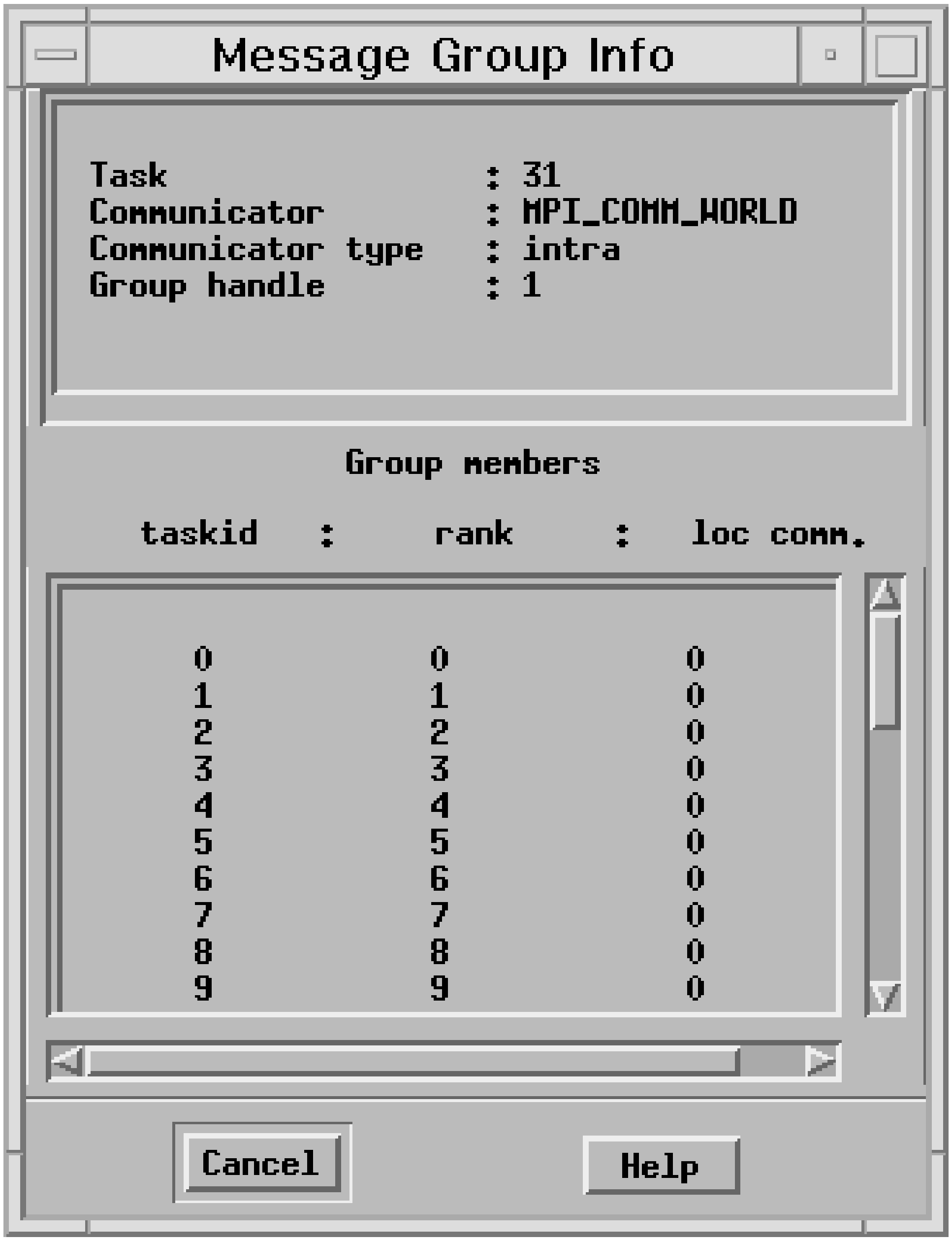
SELECT the desired task button to get the Task Message Queue window.

SELECT the desired message entry.

HOLD the right mouse button to get the Message Data menu.

SELECT **Group Info** from the Message Data menu.

This window displays information about the selected message followed by the task id, rank, and local communicator of the group members. The format of the window varies based on whether the communicator is an inter-communicator or an intra-communicator. If it is an intra-communicator, the window displays information about both the local and remote groups.



Message Group Info

```
Task                 : 31
Communicator         : MPI_COMM_WORLD
Communicator type    : intra
Group handle         : 1
```

Group members

taskid	rank	loc comm.
0	0	0
1	1	0
2	2	0
3	3	0
4	4	0
5	5	0
6	6	0
7	7	0
8	8	0
9	9	0

Figure 23. Message Group Information window

Updating Message Queue Information: Message queue information is retrieved from the task (executable) when it has been stopped by the debugger and is in **pedb** “debug” state. The task buttons at the bottom of the **pedb** main window indicate “debug” state. When the task is executed by the debugger, by stepping, etc., the message queue could potentially change. Therefore, it is necessary to update the message queue information when the task returns to “debug” state.

If the message queue debugging features are currently being used, the debugger automatically updates the message queue windows when the task returns to “debug” state after being executed. The procedure for updating is as follows:

- The Application Message Queues window is updated using the current option and filter settings.

- The tags in the Select Filters window are updated to list the current set of tags in use.
- Any Task Message Queue windows that are open for tasks in the current debugger context are updated with the current list of messages for the task. This is assuming the tasks are in “debug” state.
- Any Task Message Queue windows for tasks not in the current context, or not available, are not updated and are closed.
- Any Message Details or Message Group Information windows are closed. This is because there is no absolute way to determine if the message is still on the queue.

Visualizing Program Arrays: The Visualization window allows you to select elements of a C or Fortran array, and display the data in those elements using data visualization tools of your choice. By defining the range of elements, you can control the portion of the array that is visualized. Two types of visualizations are available with this feature: *pre-packaged* and *user-defined*.

The Visualization function is only available for arrays of integer and floating point data types.

To open the Visualization window:

PLACE the mouse cursor over an array from a local or global variable list.

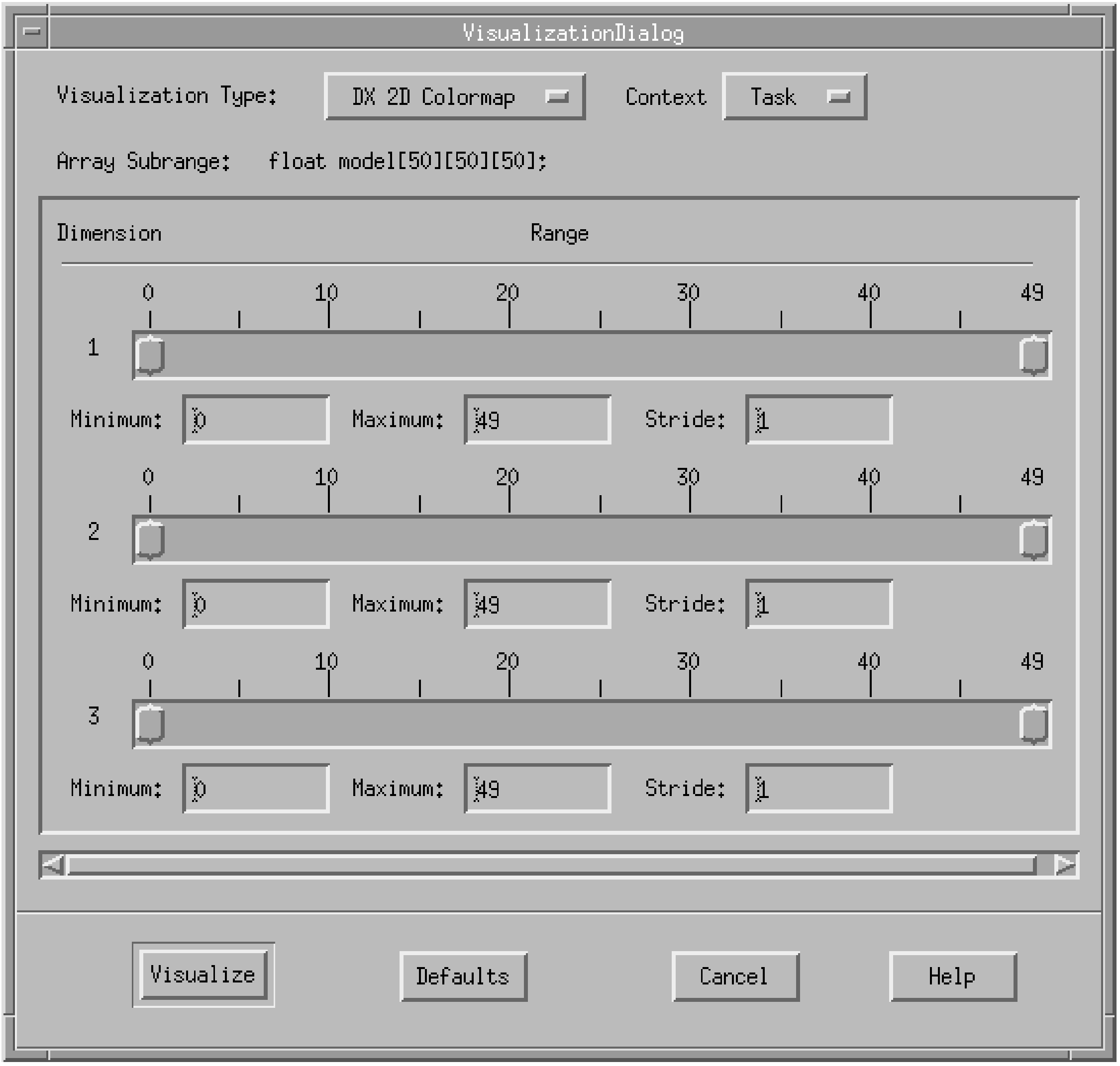
PRESS the left mouse button to highlight your selection.

PRESS the right mouse button

- A pop-up menu appears.

SELECT **Visualize...**

- The Visualization window opens, shown in Figure 24.



Visualization Type: DX 2D Colormap
Context: Task

Array Subrange: float model[50][50][50];

Minimum: 0 Maximum: 49 Stride: 1

Minimum: 0 Maximum: 49 Stride: 1

Minimum: 0 Maximum: 49 Stride: 1

Figure 24. Visualization window

Visualization Type: Clicking on the raised button in this field results in a menu being displayed to allow you to choose the method for visualizing the selected array data. The first five menu options are pre-defined visualization methods for IBM's Visualization Data Explorer product. To execute these visualizations, the Data Explorer runtime fileset is required (see Appendix D, "Visualization Customization and Data Explorer Samples" on page 243 for more information). The last five options are available for you to call user-defined visualization tools. The default value is set to "DX 2D Colormap." This menu, and the tools that can be invoked from it, are able to be customized.

The interface to the pre-defined samples is the same as that used with user-defined visualizations. Therefore, you can modify or replace them, if desired. Additional Data Explorer visualizations are included in a samples directory, but not pre-defined on the **pedb** user interface. The user-defined options are set up through the X defaults file and through a separate configuration script.

The following Data Explorer visualizations are pre-defined:

- 2D Colormap

- 2D Contour map
- Statistical analysis (text)
- 2D Height field (with a colormap)
- 3D Volume rendering.

The following are included in a samples directory:

- 3D Isosurface (3D contours)
- 3D Volume slicer.

The samples directory holds all of the visualization scripts (*.net* and *.cfg* files in Data Explorer terminology). It also includes the source for the program with which **pedb** interfaces to Data Explorer, and a makefile for the sample source.

User-defined Visualizations: You can insert your own visual tools for the debugger to invoke. When you define visualizations, you need a common method for integrating your applications.

A two step process is required to enable the tools you define and integrate with **pedb**:

1. Each menu option label has a corresponding label string entry in *Pedb.ad* (the **pedb** X defaults file). You change this string entry to your user tool name:

   ```
   Pedb*VisualTypeOption_1.labelString:  DX Colormap



   changes to



   Pedb*VisualTypeOption_1.labelString:  User Tool 1
   ```

   Note: You can edit your *.Xdefaults* file from your **$HOME** directory using the *Pedb.ad* file as a reference to the things you can change.

2. Each menu option has a corresponding Korn shell script in */usr/lpp/ppe.pedb/bin* that is executed to start the visualization:

   ```
   /usr/lpp/ppe.pedb/bin/VisualTypeOption_1.ksh
   ```

   By modifying this shell script, you can direct **pedb** to call your own tools.

Context Setting: This option menu lets you select the tasks for which array data will be visualized, based on the criteria described below. In the visualization process, the selected array data is written in HDF format to a temporary file located in /tmp. Certain context settings allow the selected array data to be written to the temporary file in separate Scientific Data Sets, one set for each participating task.

The three choices for context are:

- “Task” - visualize the array data for the task from which this Visualization window was opened.
- “Current” - visualize the array data from each task within the current context. In this context the array must be displayed in all of the Global Data area task windows in the current context for all tasks to participate.

- “All” - visualize the array data from each of the tasks that are executing in this **pedb** debugging session. This is the same as the tasks which are included in the task group “All” if you are running **pedb** in normal mode, or the task group “Attached” if you are running in attach mode. In this context, the array must be displayed in all of the Global Data area task windows for all tasks to participate.

Note: Context settings of “Current” or “All” result in multiple data sets being written to a temporary file for visualization. Verify that the tool you have chosen to perform the visualization allows you to visualize more that one data set at a time.

If the context setting is either “Current” or “All,” the following criteria must be met for a task within the specified context to participate in the data visualization:

1. An array of the same name exists on each task (within the local or global block, depending on where the variable was selected).
2. The array on that task must have the same number of dimensions as the array on the task from which the Visualization window was opened.
3. The minimum element number for each dimension of the array must match those for the array on the task where the visualization is initiated. This is only a consideration with Fortran arrays, where a program can have arrays that are declared with any integer as the minimum element number. The maximum element numbers are not checked.
4. If the array is a global array variable, then the array must be displayed in the associated task window of the Global Data area.
5. The task must be in “debug ready” state.

   If any of the tasks within the context do not meet all of the above criteria, they will be excluded from the visualization, and a message will be displayed to inform you of this.

   As stated above, the selected array data will be output to the HDF file as separate Scientific Data Sets. These data sets will be written to the file in order by task number. The visualization program is required to determine the order in which the data will be visualized.

Array Subrange Area: For details on this area, see “Specifying the Array Subrange” on page 80.

Visualize Button: Clicking on this button initiates the visualization of the selected array data. The Visualization window remains open to allow additional visualizations on this array. The visualization program will be re-initialized each time this button is pressed. This will create multiple instances of the visual, rather than refreshing the data from the previous visualization.

Stop Sign Icon: Transferring data for visualization can typically take more than a few seconds. When the data transfer begins, an icon in the shape of a stop sign appears in the upper right hand corner of the Visualization window, and remains there until the data transfer has completed. If you wish to stop a data transfer that is in progress along with the visualization, click on this icon with the left mouse button.

Defaults Button: Clicking on this button will reset all fields and states in the Visualization window back to the default settings that were used the first time this window was opened for this array on this task. The subranges for all array dimensions will be set back to their full ranges.

Cancel Button: Pressing this button closes the window without initiating the visualization. The settings and specifications from the last visualization are retained for the next time the Visualization window is opened for this array on this task. If a visualization was not performed while the window was open, the settings and specifications when the window was opened for this array on this task are retained.

Help Button: This button displays help information for the Visualization window.

Source Code Control

During a **pedb** session, the source code file displayed in the Source Area may change many times. Each time execution of the tasks in a context stops, the debugger updates the **pedb** window and checks that the source code displayed matches the program counter. For example, if execution has stopped in a procedure located in a different file, the debugger automatically updates the Source Area to display the current source.

When tasks have stopped in different source files, the lowest numbered task in debugged state in the group determines the source file displayed. This excludes tasks which are unhooked, exit requested, or exited.

To display the source from a different task, you can either change the context or open a view with a different context. You can also change the current source code displayed by opening a source code file using the Source File(s) window, selecting a line in the stack window, or double clicking on a thread in the Threads window.

Opening a Source Code File: You can open a source code file and display it in the Source Area using the Source File(s) window. To do this:

SELECT **File → Get Source File ...**

- The Source File(s) window opens.

This window contains a list of accessible source files associated with your program that have been compiled with the **-g** flag. The source path is used to find the files.

PRESS the left mouse button to select a file in the list.

PRESS **OK**

OR

DOUBLE-CLICK on the desired file in the list.

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

Source Code Search Path: The first default path searched is “ **.**” (the current directory). If you do not explicitly specify a path when choosing a file to load, **pedb** uses the AIX path established by the **PATH** environment variable to locate the file. The path in which the program is located, whether explicitly specified or not, is added to the end of the list of directories searched for source files.

You may explicitly set the source code search path on the command line when invoking **pedb** using **-I** flags. The effective search path is set to the **-I** paths specified, in the order they appear on the command line. If you do not explicitly set it, the source code search path is based on the program(s) you load for debugging in the partition, as described above.

Note: In addition to having access to your program on each remote node, **pedb** requires source files on the home node to do source level debugging. See *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for more information.

Source Path Window: During your **pedb** session, the search path used to locate source files may be modified. You may edit it, adding new paths or deleting or changing existing ones. This is helpful when source is distributed in multiple directories and you step into a source file which is located in a directory you missed at startup.

SELECT **File → Update Source Path ...**

- The Update Source Path window opens.

FOCUS on the edit field.

TYPE IN the new source search path, or modify the existing one.

PRESS OK

- The Update Source Path window closes. Subsequent source files will be accessed using the new path.

Cancel closes the Update Source Path window without changing the current source path.

Edit Current Source File: You can edit the source file which is shown in the source area by selecting the **Edit Source File** menu from the **File** pulldown menu on the main window.

PRESS **File → Edit Source File**

- You open an edit session in an aixterm window.

Notes:

1. The editor used is determined by the **$EDITOR** environment variable.
2. If the source file in the Source Area is modified using the **Edit Source File** option, the program counter icon (→) may then be out of synch. This is because the line number information is based on the compiled version of the source. If you wish to continue debugging after editing your source file, consider saving it under a different name or directory instead of overwriting the copy that the debugger is referring to.

Source File, Variable Viewer, and Threads Viewer Find: Use the **Find** option to locate text in the source code, Variable Viewer, or Threads Viewer. You first open the Find window and specify the text to find. Once you have entered text, the find options are enabled. The find options are available from the menu bar pulldown and from buttons in the Find window. Accelerators <**ctrl-f**>, <**ctrl-n**>, <**ctrl-p**>, and <**ctrl-l**> are available for **First**, **Next**, **Previous**, and **Last** respectively. Search results are displayed differently for the source code window and the Variable or Threads Viewer.

To find text in the source code window, Variable Viewer, or Threads Viewer, go to the menu bar.

PRESS **Find**

● You see a pulldown menu with the following options:

- **Open Find Dialog ...** - This option opens the Find window where you will enter the text to find.
- **First** - Finds the first occurrence of the text.
- **Next** - From the current line, finds the next occurrence of the text.
- **Previous** - From the current line, finds the previous occurrence of the text.
- **Last** - Finds the last occurrence of the text.

Using the Find Window: To open the Find window, go to the menu bar in the Main window or Variable Viewer.

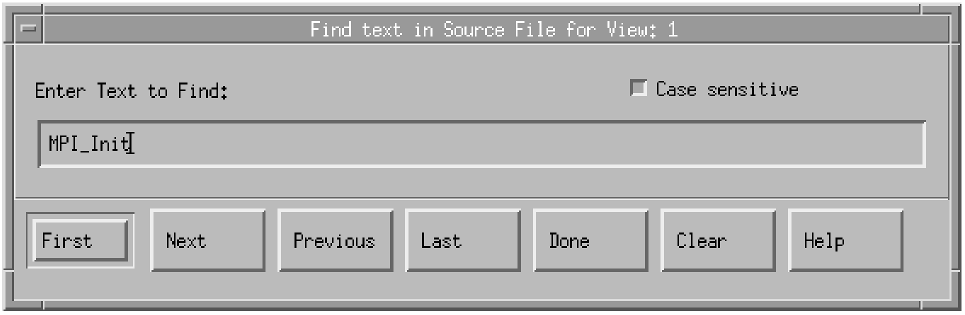
From the Main window:

SELECT **Find → Find Text in the Source Window ...**

From the Variable Viewer:

SELECT **Find → Find ...**

● The Find window opens as shown in Figure 25. The Find window title will indicate if you are using **Find** from the source code window or the Variable Viewer.


Find text in Source File for View: 1

Enter Text to Find:

MPI_Init

Figure 25. Find window

Enter the text to find in the text field. Entering text automatically enables the find option buttons in the window and the menu bar. Pressing **Enter** in the text field is the same as pressing the **Next** button. Use the **Case sensitive** toggle button to ignore the case of the text when searching. At the bottom of the Find window are the following buttons:

- Find options - These buttons (**First**, **Next**, **Previous**, and **Last**) and their corresponding buttons on the menu bar pulldown initiate a search for the text you typed.
- Done - closes the Find window.
- Clear - clears the text field. Note that the find options are desensitized (grayed out) when there is no text in the text field.
- Help - displays help text for using **Find**.

If the search fails, a message is displayed in the information area indicating the search direction and the actual string used in the search. You may want to broaden the search by specifying fewer characters or using the **Case sensitive** toggle button to ignore case.

When text is found in the source window or Threads Viewer, the entire line containing the text is highlighted. This becomes the current line, and the reference point for locating the next and previous occurrences.

When text is found in the Variable Viewer, the text that was matched is highlighted. The first character of the text is the reference point for the **Variable Options** pop-up menu as described in "Data Display Techniques" on page 76.

Source Emphasis: **pedb** provides source code emphasis when displaying code in the source area of the main window. For example, the language symbols, variable and function names, and comments are all displayed in different colors. See Table 9 below for details.

The debugger scans the current source file to identify elements of the language. Each element is then drawn with a different foreground and background color to emphasize that element. This may help you to quickly identify points of interest in your source code. It is particularly useful when you are not familiar with the code being debugged. Instead of scrutinizing the code to identify variables and comment blocks, their color will automatically alert you to their function.

To turn this feature off, set `Pedb*SourceEmphasis: False` in your *.Xdefaults* file, or use the toggle button on the menu bar:

File → Source Emphasis

The following table lists the default color scheme for each language element identified. Each resource can be given a unique color, but be aware that each unique color used will increase the total number of colors required for **pedb**. Resource values are any valid color specification for your system, for example, the values found in */usr/lpp/X11/lib/X11/rgb.txt*.

Table 9 (Page 1 of 2). Default Color Scheme

Resource	Language Element
Pedb*AlphaForeground: Pedb*AlphaBackground:	Alphanumerics - variable and functions names.
Pedb*CommentForeground: Pedb*CommentBackground:	Comments
Pedb*MessageForeground: Pedb*MessageBackground:	Message passing routines - MPL and MPI.

Pedb*KeywordForeground:
Pedb*KeywordBackground:
	Language specific keywords. For example
	`int`
	`class`
	`subroutine`

Table 9 (Page 2 of 2). Default Color Scheme

Resource	Language Element
Pedb*LiteralForeground: Pedb*LiteralBackground:	Literal (quoted) strings.
Pedb*LinenumForeground: Pedb*LinenumBackground:	Line numbers. See note 2 below.
Pedb*NumberForeground: Pedb*NumberBackground:	Numbers.

Pedb*PreprocForeground:
Pedb*PreprocBackground:
	Preprocessor directives, for example
	`#ifdef`
	`#include`
	`#pragma`

Pedb*SymbolForeground: Pedb*SymbolBackground:	Language symbols (punctuation).
Pedb*LabelForeground: Pedb*LabelBackground:	Fortran labels.
Pedb*FortranProfile:	Fortran parser profile. See note 3 below.
Pedb*ProfilePath:	Path to the Fortran parser profile. See note 3 below.

Notes:

1. This feature may not properly identify all elements of your source code in all instances. **pedb** can only scan the current source file. It is not aware of other aspects of the compilation used to produce the executable. For example, a section of code may be bracketed by the directive

   ```
   #ifdef DEBUG

   #endif
   ```

   pedb will identify the directives and the elements within the block, but cannot determine if `DEBUG` was set at compile time.

2. This feature may also be used to turn off the display of line numbers in the source code area. By setting the *Pedb*LinenumForeground:* resource to the same value as *Pedb*LinenumBackground:*.

3. For expert users, **pedb** uses a profile driven parser for the Fortran source emphasis. There are 2 profiles provided in */usr/lpp/ppe.pedb/bin/FOR.PPR* and */usr/lpp/ppe.pedb/bin/FF.PPR*. *FF.PPR* supports free form Fortran, the default *FOR.PPR* supports fixed form. Experienced users may wish to change the default profile using the *Pedb*FortranProfile:* resource, or use a local copy by changing the *Pedb*ProfilePath:*.

Other Key Features

Some other features offered by **pedb** include the capabilities of displaying multiple views, and linking to online help.

Debugging Programs Using Multiple Views: You can think of **pedb** as a *window* into the *debug space*. The window 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, the variables, and the stack trace. You can open multiple **pedb** windows and have multiple views into the same debug space.

For example, you have two tasks – tasks 0 and 1 – involved in message passing. You could open two **pedb** 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 **pedb** 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 **pedb** window to provide an additional view into the debug space:

SELECT **View** → **Open new view**

- Another **pedb** window opens.

To close a **pedb** window:

SELECT **View** → **Close this view**

Getting Help: There are help buttons on most windows and help options on many menus. Selecting these help buttons or options provide built-in online help for the particular window or menu from which the help was selected.

The **pedb** main window includes a Help button to access online help in a variety of ways by selecting one of the following options:

Help on Main Window: Displays built-in online help information about the **pedb** main window. There are main window left and right button presses that may not be obvious. Here you will find a list of the actions available using the buttons on the main window.

Help on Main Window Menu Bar: Displays built-in online help for the main window menu bar pop-up menus: **File**, **View**, **Group**, **Find**, and **Options**.

Index of Online Help Topics: Displays a list of the built-in online help items that are available on the various windows and menus throughout the **pedb** debugger. Any of the listed items can be selected, which results in a window displaying that help section.

Notes:

1. The main window menu bar pull downs (**File**, **View**, **Group**, **Find**) do not have help options. You can get help about options by pressing the Help button in the upper right corner of the main window, then selecting the **Help on Main Menu Bar** option.
2. The menu bar pull downs in the Local or Global Variable windows do not have help options. You can get help about these options by pressing the Help button in the upper right hand corner of the Local or Global Variable windows.

Customizing pedb Resources: Customizable resources for **pedb** are defined in */usr/lpp/ppe.pedb/defaults/Pedb.ad* (the **pedb** X defaults file). In this file is a set of X resources for defining graphical user interfaces based on the following criteria:

- Window geometry
- Push button and label text
- Pixmaps.

Leaving pedb: It is possible to end the debug session at any time using either the **Quit** option, or the **Detach** option if debugging in attach mode.

To end a debug session in normal mode:

SELECT **File** → **Quit** from the **pedb** Main Window.

● The Quit Confirmation window appears.

PLACE the mouse cursor over **OK**.

PRESS the left mouse button.

● The **pedb** window closes and you return to the window from which you started **pedb**.

To end a debug session in attach mode, you can choose either **Quit** or **Detach**. Quitting causes the debugger and all the members of the original **poe** application partition to exit. Detaching causes only the debugger to exit and leaves all the tasks running.

SELECT **File** → **Quit** from the **pedb** Main Window.

● The Quit Confirmation window appears.

PLACE the mouse cursor over **OK**.

PRESS the left mouse button.

● The debugger session ends, along with the **poe** application partition tasks.

OR

SELECT **File** → **Detach** from the **pedb** Main Window.

● The Detach Confirmation window appears.

PLACE the mouse cursor over **Detach**.

PRESS the left mouse button.

● The debugger session ends. All tasks have exited, but stay running.

Clicking on this button causes **pedb** to exit, and allows the program to which you had attached to continue execution if it hasn't already finished. If this program has finished execution, and is part of a series of job steps, then detaching allows the next job step to be executed.

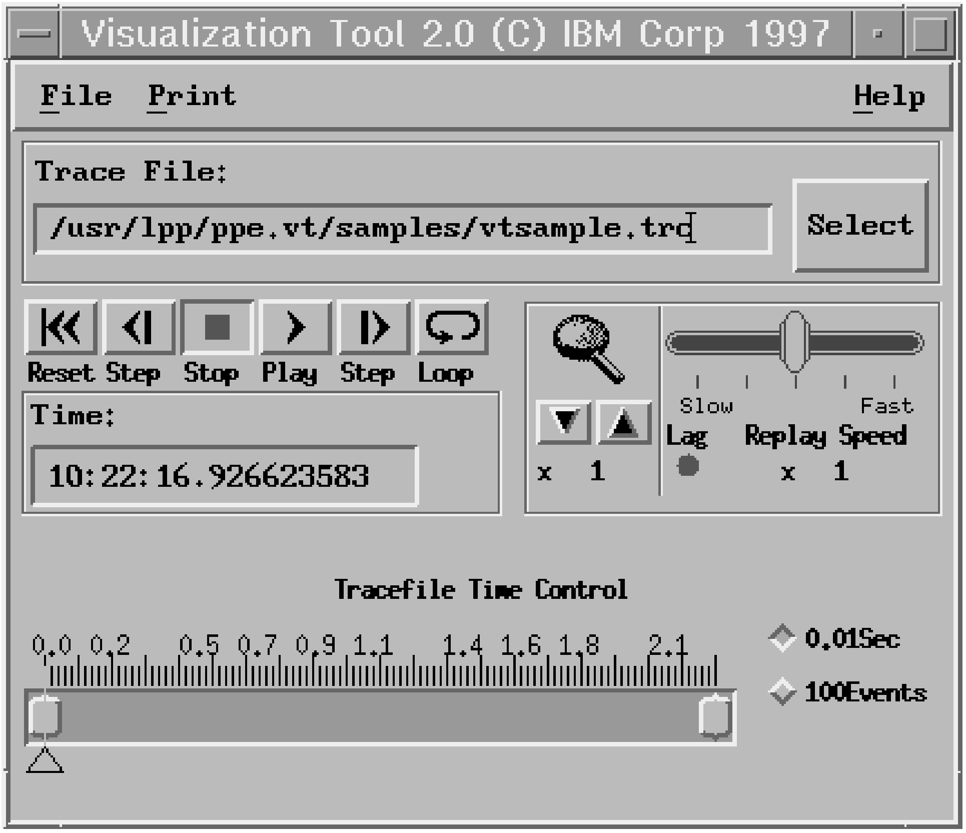
If instead you want to exit the debugger and end the program, cancel the Detach Confirmation window and use the **Quit** option as described above.

Chapter 3. 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 117 describes how to run a program to generate files containing trace records, and how to play back these trace records using VT.



Trace File:
/usr/lpp/ppe.vt/samples/vtsample.trc

Time:
10:22:16.926623583

Figure 26. 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 IBM RS/6000 SP or RS/6000 network cluster. In performance monitoring, VT only displays system statistics and not communication information. See "Using VT for Performance Monitoring" on page 139.

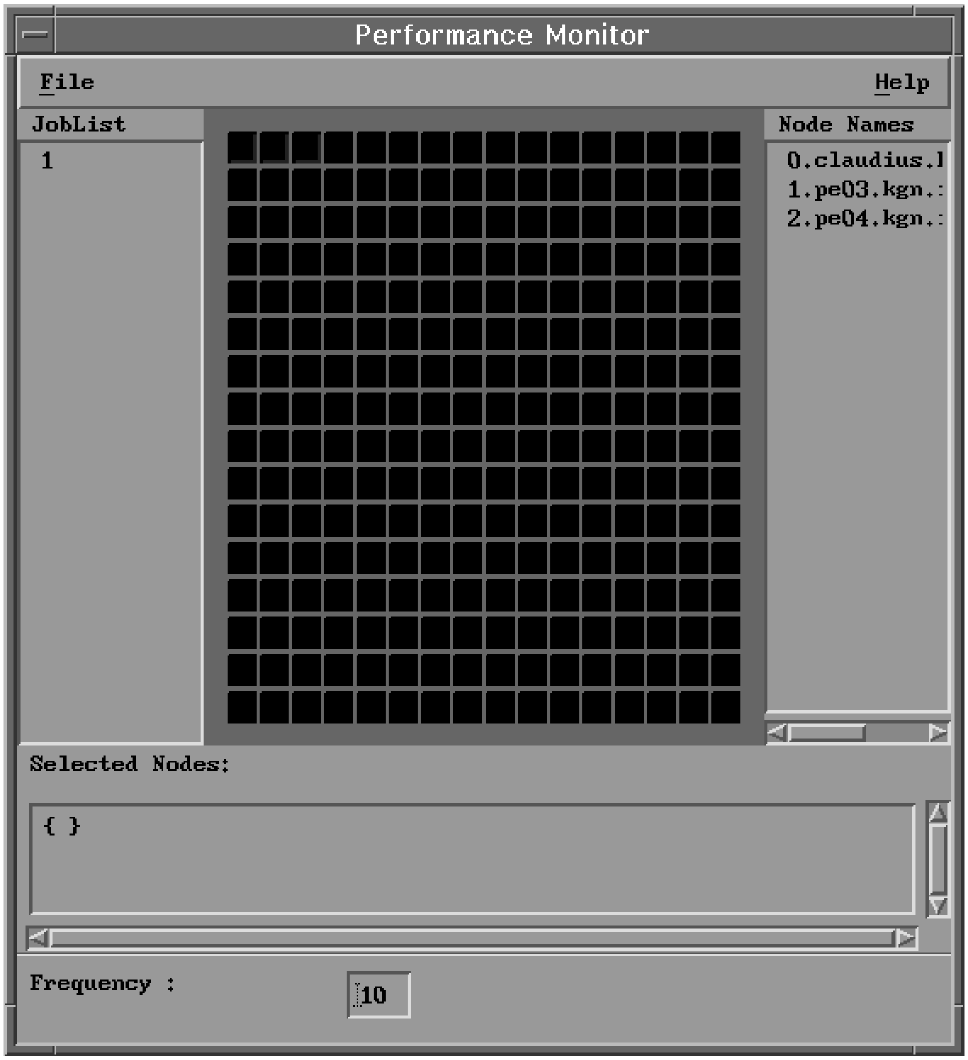


Figure 27. VT Window for Performance Monitoring

The VT views enable you to visualize:

- communication among processor nodes (only available in trace visualization mode)
- the type and duration of communication events (only available in trace visualization mode)
- the size of the messages (only available in trace visualization mode)
- CPU utilization of processor nodes
- a parallel program's source code as it relates to the executable's run (only available in trace visualization mode)
- the number of times 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 114 contains a general description of views and their use. “View Descriptions” on page 148 contains a more detailed description of the views, the information each presents, and how to interpret them.

Table 10 on page 112 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 113.

Note: For additional information on VT, refer to *IBM Parallel Environment for AIX: Hitchhiker's Guide*

Table 10. VT Quick Operation

For Trace Visualization:	**For Performance Monitoring:**
• 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 “Step 2: Compile the Program” on page 120. For information on compiling threaded C, C++, or Fortran programs using **mpcc_r**, **mpCC_r**, or **mpxlf_r**, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* • 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/ppe.vt/samples/vtsample.trc* ● The Trace Visualization window and the View Selector window open. • 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 148. • 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**	• Step 1: Start a VT session for performance monitoring. If you are in an environment where the SP system Resource Manager is available: **ENTER** **vt** If you are in an environment where the Resource Manager is not available: **ENTER** **vt -norm** ● The Trace Visualization window and the View Selector window open. **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 SP system or cluster. The processor nodes are also listed by name in the Node Name List, and, if you are using an SP 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 148. • 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.

Table 11. Starting VT

To start VT for trace visualization:	To start VT for performance monitoring when the Resource Manager is available:	To start VT for performance monitoring when the 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 117.	**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 139.	**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 139.

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 12 (Page 1 of 2). 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 on page 128
“Step 1: Load a Trace File for Playback” on page 127			
-go	Start playing back the trace file immediately upon starting VT. When you use this flag, you must also specify a configuration file using the **-configfile** (or **-cfile**) flags.	**vt-cfile** *config* **-go**	page 133
“Step 2: Start Playback of a Trace File” on page 133			
-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 148

Table 12 (Page 2 of 2). VT Command-line Flags

Use this command-line flag:	**To:**	**For example:**	**For more information, see:**
"Saving and Loading a VT Configuration File" on page 147			
-cmap	Request a private color map. When you use this flag, VT's color allocation is independent of other X-Windows applications.	**vt -cmap**	page 147
"Adjusting a View's Time Resolution and Colors" on page 144			
-norm	Indicate that you are not using the Resource Manager. If you are using an RS/6000 network cluster, you must use this option.	**vt -norm**	table 11
-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 159
"Adding and Deleting Paths to the Source" on page 159			
-log_file	Specify the file name where the results of the trace file post-processing will be written. The default name is **$HOME/tracefilename.pplog**.	**vt -log_file** *logfile*	page 128
-h **-?** or **-help**	Get help.	**vt -h** **vt -?** or **vt -help**	page 138
-mp_source	Specify which task's source code is displayed in the Source Code view.	**vt -mp_source 2**	page 157

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 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: Those views displaying cumulative information are only valid from the point at which the display was opened. Opening views after playback has started will result in incorrect cumulative information. See “View Descriptions” on page 148 for more information on cumulative views.

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.

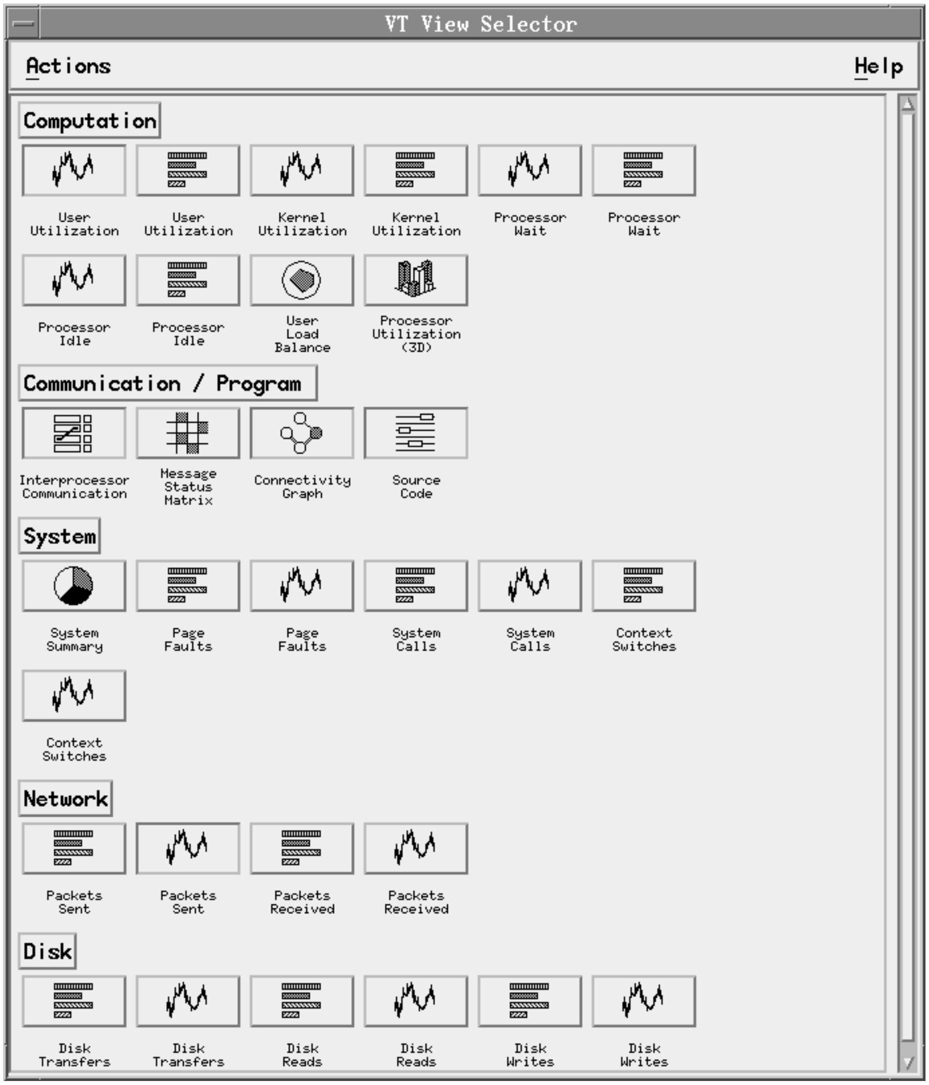


Figure 28. View Selector Window

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 148 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.

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.

Note: If you are running the Parallel Environment on an SP system with the High-Performance Communication Adapter configured, ensure that the switch fault handler (fault_service_Worm_RTG) is running before using the trace facility of the Visualization Tool. The digd daemon requires services that are initialized by the fault handler. The switch fault handler and the digd daemon are specified in */etc/inittab* and are started automatically. If the digd daemon is used by the trace facility before the switch handler is running, a switch channel check may occur which could cause the switch to crash.

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* and *VT_mpi.h* in the directory */usr/lpp/ppe.poe/include* defines the structure of trace records. Refer to this header file if you want to write your own program to read, manipulate, or interpret trace records independent of VT. A sample program is provided in */usr/lpp/ppe.vt/util/rtrc.c* which displays the contents of a trace file in ASCII format. This program can be used as an example for creating your own program to extract information from the trace file. Refer to “Trace File Post-Processing” on page 128 for more details on post-processing, and refer to the appropriate VT README file for more details on the format and structure of the individual trace records.

Message Passing

Message passing trace records contain information regarding point-to-point 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 a message passing subroutine. Message passing subroutines are described in greater detail in *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference* and *IBM Parallel Environment for AIX: MPL Programming and 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 a collective communication subroutine, for example: **mpc_bcast** or **MPI_Bcast**. Collective communication subroutines are described in greater detail in *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*

AIX Kernel Statistics

AIX kernel statistics 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 *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* 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 157. The mpc_marker and MP_MARKER Parallel Utility Function calls are described in *IBM Parallel Environment for AIX: MPI Programming and Subroutine Reference*

Trace Record Timestamps

If the High-Performance Communication Adapter is configured, the trace records are timestamped with the switch clock value, regardless of whether the adapter is used for communication. If the adapter is not present, the system clock is used. When the tracing process completes, the trace records are ordered by timestamp and the switch clock values are converted to a time of day timestamp.

The tracing routines use the synchronized counter on the communication adapter. When tracing is initialized, **VT_trc_init()** synchronizes the trace file timestamp of all tasks to the time of task 0. During the run, trace records will only use the register as a timestamp. When the application on a single node is complete, **VT_trc_done()** is called, which merges the communication and kernel statistics records into a single file. It also maps the counter timestamp to the time of day using the synchronized timestamp from **VT_trc_init()**.

When no communication adapter is present (for example when running on a cluster of RS/6000 workstations), the Time of Day (TOD) clock provides the timestamp information. The TOD clock should be synchronized across all nodes so that the timestamp information can be properly correlated, otherwise the system times may produce misleading results. For example, a receive may appear to complete before the matching send. This is obviously not possible, yet if the system clock of the sender is behind the clock of the receiver, the trace record will be timestamped, and therefore visualized as if it occurred later in time. The TOD can be set on multiple nodes using a standard synchronization tool like Network Time Protocol (NTP).

Generating Trace Files

To generate trace files:

1. VT generates trace records for all events for the entire duration of the program. 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 flag when you invoke your program. See “Step 3: Process the Program to Generate a Trace File” on page 120 for more information.

2. Compile your program as described in “Step 2: Compile the Program” on page 120.

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 120.

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
`CALL VT_TRC_STOP ( INTEGER RETURN_CODE )`	`int VT_trc_stop_c()`
`CALL VT_TRC_START ( INTEGER FLAGS, INTEGER RETURN_CODE )`	`int VT_trc_start_c( int flags )`

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

Table 13. Trace Record Integer Flags

Flag Value	Message Passing	Collective Communication	AIX Kernel Statistic	Application Markers
0				
1				√
2			√	√
3	&check	√		√
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
`CALL VT_TRC_STOP ( RC )`	`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
`CALL VT_TRC_START ( 3, RC )`	`VT_trc_start_c(3)`

Note: 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 134.

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** compilers. 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 *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*
2. For information on compiling threaded C, C++, or Fortran programs using **mpcc_r**, **mpCC_r**, or **mpxlf_r**, see *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*
3. For more information on the **-g** option, refer to its use on the **cc** command as described in *IBM AIX Version 4 Commands Reference*

Step 3: Process the Program to Generate a Trace File

To generate a trace file, you need to run 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 most 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.

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 13 on page 119. 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 enable tracing with **MP_TRACELEVEL**, you may then run tracing on and off within the program using *VT_trc_stop* and *VT_trc_start*. If you do not enable tracing with **MP_TRACELEVEL**, calling *VT_trc_stop* and *VT_trc_start* within the program will have no effect.
2. Useful informational messages can be displayed 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 *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for more information on **MP_INFOLEVEL**.

Specifying a Trace File Name: By default, trace files are named 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 to temporarily override their associated environment variable.

For example, say you generate a trace file for *program*. If you do not specify a name, it is named *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 most 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 Parallel Environment for AIX: MPI Programming and Subroutine Reference* for more information.

Managing Storage for Trace Files: The system uses the following three-tiered approach to manage the storage of trace files:

1. The system writes the trace records to a 1 MB buffer in memory on the local nodes.
2. When the memory buffer is full, the records are appended to a file in the directory specified by the **MP_TMPDIR** environment variable or **-tmdir** command line parameter. It is suggested that the temporary directory be a directory that is local on each node so that network traffic is minimized.
3. The files from all the nodes are merged into a single file in the directory specified by the **MP_TRACEDIR** environment variable. **MP_TRACEDIR** can be overridden by the **-tracedir** command line option. Refer to *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for more information.

Communication event records and system statistics trace records are written independently to files with generated temporary names. Communication trace records are written by instrumentation in the communication library. System statistics are written by a spawned process which samples the kernel at a specified interval.

To reduce its impact on an application's performance characteristics, the tracing process makes efficient use of memory, disk I/O, and network traffic during execution and defers formatting, synchronization, and derivation operations until the application is complete. The three-tier tracing architecture passes large data blocks to the disk at less frequent intervals rather than passing small data blocks continuously. In addition, the format of the trace records generated on the local nodes while the job is running is more compact than the format of the records read during trace playback so that it can be generated more quickly.

Integration of the trace data from the individual nodes into the final trace file on the home nodes is deferred until the application completes. VT uses an internal communications channel provided by POE to distribute clock synchronization information and collect trace data. Figure 29 on page 123 illustrates the three-tiered approach used to manage the storage of trace files.

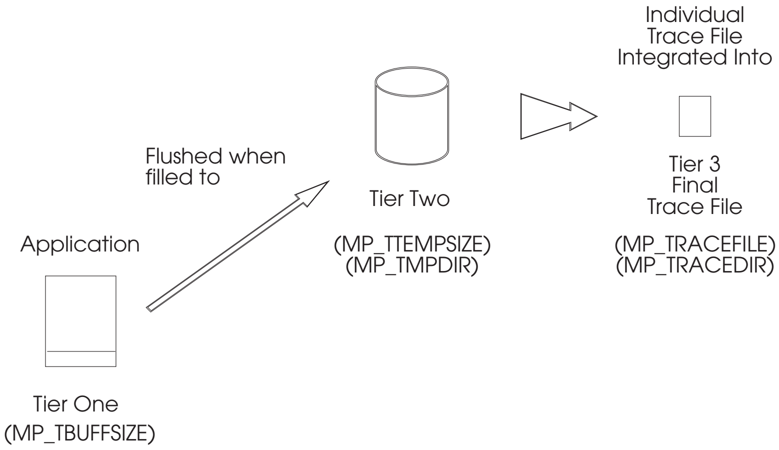


Figure 29. Three-Tiered Trace File Storage Approach

You have several options regarding the storage of trace files. You can specify:

- a directory other than */tmp* for the temporary trace file.
- a directory other than your current directory for final trace file output.
- the maximum size of the buffer and the temp 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 in */tmp*. 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 most POE command-line flags, the **-tmpdir** flag temporarily overrides its associate environment variable.

For example, to use a *samples* directory under your home directory for temporary trace files, you could:

Set the MP_TMPDIR environment variable:	Use the -tmpdir flag when invoking the program:
ENTER **export MP_TMPDIR=***$HOME/samples*	**ENTER** **poe** *program* **-tlevel** *9* **-tmpdir** *$HOME/samples*

Writing the Final Trace File to a Specified Directory: By default, VT writes the final trace file to the your current directory. You can use the environment variable **MP_TRACEDIR** to specify a different directory. The directory specified by **MP_TRACEDIR** must be accessible to all the nodes of a partition. If a partition consists of more than one node, the trace file must not reside in a local 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.

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 Parallel Environment for AIX: MPI Programming and Subroutine Reference* for more information.

Say you wanted to increase the size of the buffer to 5 MB, and the size of the temp files to 20 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*	**ENTER** **poe** *program* **-tlevel** *9* **-tbuffsize** *5M* **-ttempsize** *20M* or **poe** *program* **-tlevel** *9* **-tbsize** *5M* **-ttsize** *20M*

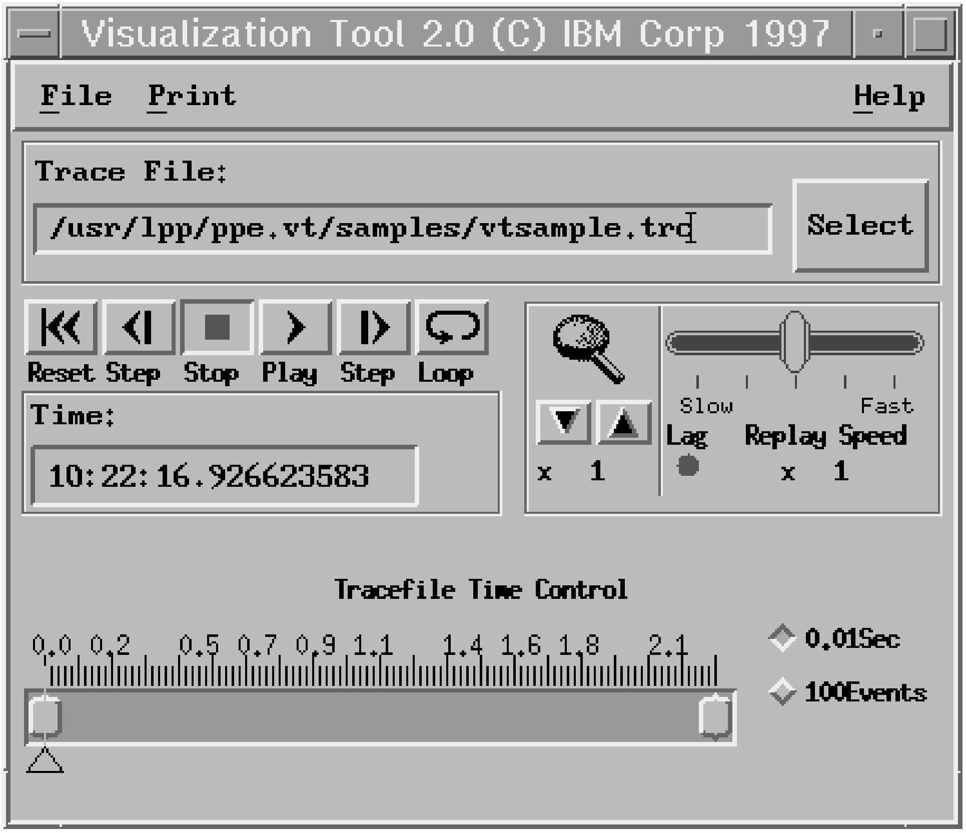
Note: Remember that specifying a larger buffer size decreases the amount of free memory available to the application.

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 most 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.



Visualization Tool 2.0 (C) IBM Corp 1997

Trace File: /usr/lpp/ppe.vt/samples/vtsample.trc

Time: 10:22:16.926623583

Figure 30. 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
  - Print
  - Help

  These menus may also be posted to a separate window ("torn off") by selecting the dashed line below the menu title when the menu is first posted.

- The *Trace File Field*. This field is just below the Menu Bar. It displays the full path name of the trace file you have selected for playback. You can also select a file for playback by typing it in this field and then pressing **Enter**.

- The *File Select Button*. This manages the File Selection box as described in "Step 1: Load a Trace File for Playback" on page 127.

- 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 Trace File Field 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 *Magnification Control*. This control (to the right of the Basic Trace Visualization Controls) 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 136 for more information on this control.
- The *Replay Speed Control*. This control is located to the right of the Magnification Control. 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. For more information on speed control, see “Step 6: Adjust Playing Speed” on page 135.
- The *Application Time Display*. This display (just below the Basic Trace Visualization Controls) 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 119), the time shown in the Application Time Display will skip over the time during which trace record generation was set off.

  The Application Time Display is an editable field. You can click on the time field and manually enter a time. This allows you to pinpoint a time in the trace file for a particular view. See “Step 9: Going to a Specific Time” on page 136 for more information.
- 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 the scale is elapsed time or number of trace records. 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 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”
start trace file playback	“Step 2: Start Playback of a Trace File” on page 133
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 133
cycle playback through a trace file or a portion of a trace file	“Step 4: Cycle Playback Through a Trace File” on page 134
reverse or advance stepping by a single trace record	“Step 5: Stepping Playback” on page 134
adjust the speed of playback	“Step 6: Adjust Playing Speed” on page 135
scroll back over view history buffers	“Step 7: Scrolling Over History Buffers” on page 136
adjust the level of detail shown in certain views	“Step 8: Adjust View Magnification” on page 136
move to a place in the trace file by entering a time directly	“Step 9: Going to a Specific Time” on page 136
print a view	“Step 10: Printing a View” on page 137
get help	“Step 11: Getting Help” on page 138

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/ppe.vt/samples*. 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 114. 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:

PRESS the **Select** button on the VT control panel.

OR

SELECT **File → Tracefile → Select**

- 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
- A list showing up to the last 10 trace files that were viewed.

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 one of the lists 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/ppe.vt/samples*. You would:

  ENTER **vt -tracefile** */usr/lpp/ppe.vt/samples/vtsample.trc*

  or

  vt -tfile */usr/lpp/ppe.vt/samples/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.

Trace File Post-Processing: The visualization tool relies on instrumentation in the communications library for matching and eventually visualizing communications events. That is, each point-to-point and collective communications event must be paired to the corresponding events. For example, a send must be paired to the matching receive, and all members of a broadcast must be linked together for visualization. Intermediate files from each node contain part of this information, and after the files are merged into the final trace file a post-processing phase is necessary to do this pairing and generate summary information. The post-processing rewrites parts of the trace records and permanently changes the file, so it is required only once and may be subsequently replayed any number of times. Any errors detected during the post-processing are recorded in the post-processing log file named *$HOME/<trace file name>.pplog*.

If this file cannot be opened for writing, the output will go to standard error (stderr). An example of the contents of this file follows:

```
VT Post-processing: Processing of trace file /tmp/sr.trc begins.

VT Post-processing: Program source file sr not found.

No source lines will be displayed.

VT Post-processing: Largest single message sent was 64 bytes from task 1

VT Post-processing: Largest cumulative messages sent was c8 bytes

between tasks 0 and 1

VT Post-processing: Total of 35 communications events processed

Trace file type: Post-Processed (5)

Trace file release level: 0203

Time of first trace event: 17:01:53.199213774

Time of last trace event: 17:01:53.676011400

@(#) IBM Visualization Tool trace file, Release :  0203

Number of processors used: 0004

Number of unmatched events in file: 00000000

Maximum single transfer: 00000064

Maximum cumulative transfers: 000000c8

VT Post-processing: Processing complete.
```

The log begins with the name of the trace file being processed. The next line indicates that the executable which produced this trace file could not be found in the current PATH. If available, post-processing will use the internal debug information in the executable (XCOFF) to print source code line numbers whenever an error is found. In this case, no post-processing errors are reported, so the absence of the executable is not a problem. However, you will not be able to use the Source Code display in VT. The next three lines are a summary of the message size information collected by post-processing. This includes:

1. The largest single message

   This is the largest single message sent between any two processes. In this case it was 64 (hex) bytes sent from task 1 to task 0.

2. The largest cumulative messages sent

   Each message sent between any two tasks is recorded in an array of <*number_of_tasks*> squared. The size of each message from one task to another is recorded in the row of the sending task and the column of the receiving task. All messages are counted, including point-to-point and collective communications. At the end of post-processing, this array is scanned to identify the largest cumulative amount of data exchanged between any two tasks (identified by the row and column of the array). This information is used by the Message Matrix display to visualize message size information.

3. Number of communications events processed

   AIX statistics records are ignored by post-processing. This is the number of trace records collected for all the communications events.

The remaining lines on the log file print out values from the trace header record (as described in *VT_trc.h*). This includes:

- type (in this case **VT_MATCHED_TRACEFILE** or 5)
- trace_rel
- info_string
- timestamp of the header record - This is set to the timestamp of the first trace record captured.
- end_time - This is set to the timestamp of the last trace record captured.
- num_procs (number of tasks)
- unmatched_comm_cnt (all were matched successfully in this case)
- max_single_message
- max_cumulative_message

Following is an example of output with some error information:

```
VT Post-processing: Processing of trace file /tmp/sendrecv_rep.trc

begins

VT Post-processing: Program source file name is sendrecv_rep.c

VT Post-processing: Warning MPI Operation with non zero return code on

task 3 at time 10:54:39.543372800 (trace file offset 3ec)

From line 35 of sendrecv_rep.c

VT Post-processing: Warning Unmatched Point-to-Point Event on task 0

at time 10:54:40.582377000 (trace file offset 59a)

From line 17 of sendrecv_rep.c

VT Post-processing: Largest single message sent was 64 bytes from

task 1 to 0

(the rest of the summary lines are omitted)
```

In this example, post-processing was able to find the executable file; and it was compiled with **-g** so source lines could be displayed.

Note: The executable should be in the current PATH and if you plan to use the Source Code Display, you should also have access to the source code in the current working directory.

There are 2 error messages displayed. The first message indicates that there was a non-zero return code from an MPI operation and the task on which it occurred. This return code may not be fatal and should be defined on the MPI Standard. The error message also calls out the time and the source code line number of the call to the routine. To better isolate this problem, load the trace file into VT and open the Source Code Display. Move to the time reported in the message by entering it in the time field, moving the slider on the time gauge or playing and stepping up to the event (all of these operations are described throughout this chapter). The Source Code Display should now be displaying the source code line under the colored bar for task 3. VT only captures the return code, and your program may or may not continue. If it was terminated this will be obvious with VT because there will be no more activity on that task. You will probably want to determine the exact meaning of the return code and update your source appropriately.

The second error message indicates that the post-processing was unable to match a point-to-point communications event. As mentioned earlier, instrumentation in the communications library provides data whereby the post-processing should be able to correlate all messages from their source to their destination. There are several reasons why this could have failed:

- *Tracing was off at the time the message was sent or received*

  Tracing may have been explicitly turned off in the application with the **VT_trc_stop()** function, or may have been disabled because of some error condition (for example, **MP_TEMPSIZE** was exceeded). In this case, the communication may have completed successfully, but post-processing has no way of knowing this because there was no trace record captured.

- *Application program error*

  The application program may have specified an invalid destination task or some other parameter error which, although not fatal, resulted in no message being sent.

- *Problems during integration of node trace files*

  At the end of the application program, the temporary trace files from all the nodes are merged into the final trace file on the home node. If an application program is terminated abnormally (for example, <**Ctrl-c**>) or if there is an error integrating the node trace files (for example, ran out of space), then one or more of the node trace files may be incomplete or missing. In this case, there is more than likely many error messages from the Parallel Environment indicating what went wrong.

When unmatched errors occur, first make sure that the **poe** job completed successfully (no core dumps, interrupted tasks, error messages, etc.). If this is the case, then follow the steps above to play the trace up to the time listed in the error message. Then, check the information in the Interprocessor Communications and Source Code displays to isolate the problem. If you are using a MPMD (Multiple Program - Multiple Data) program, you will need to read the next section Using VT with MPMD applications to specify which program will be displayed in the Source Code display.

There are three more possible error conditions that could be reported. They are:

1. Unmatched Collective Communications Event

   One of the tasks that should have been involved with the collective communications was not found. This could occur for the same reasons as the unmatched point-to-point event (a terminated task or tracing off, for example). If your program produced the correct results, this may be a trace collection problem and the visualization will simply ignore that member of the group.

2. MPI Operation Cancelled

   MPI allows pending communication events to be canceled by the user application. When the cancellation is successful, a flag is set in the trace record and this message is displayed when the record is processed. Use the Source Code and Interprocessor Communications displays to identify the operation that was canceled.

3. Incomplete group info record detected. Continuing.

   This is an error during the capture of group information for a collective communications event. The specific group definition is terminated at the last valid record processed. Any tasks that were omitted will not be visualized as part of the group. In most cases this will occur because of a failure in trace generation or integration and will be accompanied by other error messages from the Parallel Environment. Regenerate your trace file and start VT again.

Use the steps listed above to verify that the application and the Parallel Environment completed without errors, then use VT to isolate the specific area of failure.

Using VT with MPMD applications: MPMD applications use different application programs on each node. VT only supports one Source Code display and by default it uses the program information from task 0. During playback, nodes that are not running the same executable may erroneously update in the Source Code display. In order to better support MPMD programs, a new command line flag has been added to allow you to specify which task, and therefore which executable program and source code will be shown in the Source Code display.

To specify a specific task for the Source Code display, use the **-mp_source** flag on the VT command line followed by the task number. Specifying an invalid task will cause VT to default to task 0. For example, if you are running an MPMD application as follows:

task	running
0	server_program
1	sub_server_program
2	client_program
3	client_program

and post-processing indicates an error on task 2 (in the `client_program`), you could display the source for `client_program` in the Source Code display by starting VT with **-mp_source 2**.

Note: Remember that in this case, only tasks 2 and 3 will update correctly in the Source Code display.

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 114. 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 configuration file using the **-configfile** or **-cfile** flags. See "Saving and Loading a VT Configuration File" on page 147 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 trace file:

PRESS the Loop Control Button.

- When VT reaches the end of the trace file, it returns to the start of the trace file 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 119. 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. You can interrupt this read by pressing the stop button.

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

Note: You may also step over multiple events by pressing and holding the Step Button.

Step 6: Adjust Playing Speed

During normal playback, VT attempts to play the trace file at the same rate as it was collected. 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. VT attempts to playback trace records at the same rate that they were collected In other words, if one trace record was captured every second, VT would display one every second so that the playback time would be similar to the execution time of the parallel application. If the pace of events is too fast for VT to display, it is indicated by the lag light on the lower left of the playback speed control. All events are still displayed; the lag light merely indicates that the display of events is lagging behind the actual elapsed time during collection.
2. You can also change the update rate, or time resolution, for individual views as described in “Adjusting a View's Time Resolution and Colors” on page 144.

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 back from the point at which Stop was pressed.

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. The Kernel Utilization (Graph) view (see Figure 38 on page 164) is an example of 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.

Step 9: Going to a Specific Time

With the editable field of the Application Time Display, you can move to a new time within the trace file by specifying hours, minutes, seconds, and fractions of seconds (`h:m:s.fractions`). If the time is preceded by a plus or minus sign (+, –), the value taken is relevant to the current time. When typing in a number this way, the default value is seconds. Entering + 1 will move the time forward one second. On the Trace Visualization window:

FOCUS the cursor over the Application Time Display field.

PRESS the left mouse button.

● This activates the cursor and allows you to type in a value.

ENTER the new value.

PRESS **Enter**

● The views are updated to the new time entered.

Note: If there is no event at the exact time specified, playback is positioned at the closest earlier event.

Step 10: Printing a View

Print allows you to capture images from the screen and format them to a PostScript file. The image can be printed directly or saved to a file for printing or viewing with a PostScript viewer. VT will optionally annotate the output with the following information:

- Print file creation date
- Trace file name
- Trace file creation date
- Trace file current time
- VT display name.

The Print Options Dialog lets you choose the way you want to print your views. From the menu bar of the Trace Visualization window:

SELECT **Print → Options**

● The Set Print Options window opens. From this window, you can select the following with regard to PostScript output format:

- File - sends the output to a file.
- Printer - sends the output to a printer.
- GreyShades - allows you to specify grey shading in numbers of 2, 4, or 16.
- Color - specifies color output.
- Annotate - sets on or off the display of file information, including file name, and time and date of file creation and printing.
- Delay before Grab - allows you to set a specific time within which you can rearrange your windows before the cursor activates to “grab” a view.
- Enable Landscape - creates landscape formatted output.
- Default Directory - if you select “Output to File,” this is where the file is created.
- Print Command - if you select “Output to Printer,” this is the command that is executed.

PRESS **OK**

● The Set Print Options window closes and VT sets the print options you chose.

To print all the displayed views:

SELECT **Print → All**

- All the displayed views are selected for printing. This option captures each one of the views individually, and assigns a different name for each PostScript file.

To print one selected view:

SELECT **Print → Select**

- The cursor for window selection changes to a “hand” after the delay that you specified with the “Delay before grab” option.

FOCUS on the window you want to print.

PRESS the left mouse button.

- The window is selected for printing. This option creates the PostScript file named **Vt.printRequest.ps** in the specified directory.

Note: If the PostScript file already exists, it will be overwritten.

Pressing the right mouse button cancels the print operation.

Printing the selected view in this way captures the entire window and Motif borders. You can also capture and print a rectangular area anywhere on the screen.

FOCUS the cursor on the area you want to capture.

PRESS the middle mouse button.

- This establishes one corner of the rectangle and allows you to drag a “rubberband” outline of the area.

RELEASE the button to establish the second corner.

- The cursor changes back to a “hand” and you can do one of three things:

  1. Re-establish the corners by pressing the middle button and dragging to the new location.
  2. Relocate the entire rectangle by pressing the middle mouse button within the rectangle.
  3. Print the contents of the rectangle by pressing the left mouse button.

Step 11: Getting Help

VT provides help information by starting Netscape with specific search information which will place the user at the article of interest.

Note: To access VT online help, the **ppe.pedocs** file set must be installed first. Refer to *IBM Parallel Environment for AIX: Installation* for more information.

You may get help for VT in one of three ways:

1. Pressing the **Help** button from a VT window.
2. Selecting context sensitive help from the Playback Control window:

   SELECT **Help → Context**

   - The cursor is changed to a **?**.

FOCUS the cursor over the item of interest in any VT window.

PRESS the left mouse button.

- The help information is displayed.

3. Selecting **Help** → **About** from the Playback Control window. An information pop-up displays the date and time when the VT was created.

The context sensitive help is provided by invoking the following shell script:

```
/usr/lpp/ppe.vt/bin/invokehtml.sh -display host:screen tag_name
```

where `-display host:screen` displays the help on the same VT screen, and `tag_name` is the name tag used in the HTML file.

This shell script first checks the availability of the file **/usr/lpp/ppe.pedocs/peopsuse2.html**, and if Netscape is available. Please see the shell script for customization instructions.

Note: There may be a long delay when Netscape is first invoked. Any subsequent invocation will not experience the same delay.

The default shell scripts can be overridden using the **MP_VT_HELP_SHELL** environment variable to point to your own shell script. This allows you to change the default browser or HTML file without involvement of the system administrator.

Using VT for Performance Monitoring

Performance monitoring enables you to monitor the operational status and activity of any of the processor nodes of your SP system or RS/6000 network cluster. You do not need a trace file for this, because VT does the monitoring in real time. 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.

Nodes that you select for monitoring are specified to VT in one of three ways:

- VT can use the Resource Manager (RM) to identify nodes to be monitored. The RM allocates the nodes available to run jobs. These nodes are specified for dedicated or shared use in the host list file you create. Refer to *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for more information on the RM and allocating nodes.
- You can supply VT with a specific list of nodes.
- VT can query all the nodes on the LAN, looking for the Statistics Collector daemon.

The default is for VT to contact the RM for the nodes it currently manages. In this mode, VT is aware of the parallel jobs the RM is controlling and can show allocation of jobs to nodes.

If the RM is not present, you should use the **-norm** option when starting VT to tell it not to try and connect with the RM. In this case, VT will start with the host list file you create, containing the list of nodes on which to run jobs, and will add to the list any other nodes on the LAN that are running the Statistics Collector daemon. Although VT can still monitor nodes, it does not know about parallel jobs or allocation of jobs to nodes.

Note: If you are running the Parallel Environment on an SP system with the High-Performance Communication adapter configured, ensure that the switch fault handler (fault_service_Worm_RTG) is running before using the performance monitoring function of the Visualization Tool. This function uses the digd daemon. The switch fault handler and the digd daemon are specified in */etc/inittab* and are started automatically. If the digd daemon is used by the performance monitoring function before the switch handler is running, a switch channel check may occur which could cause the switch to crash.

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.

Note: The playback of trace files is disabled when using the Performance Monitor.

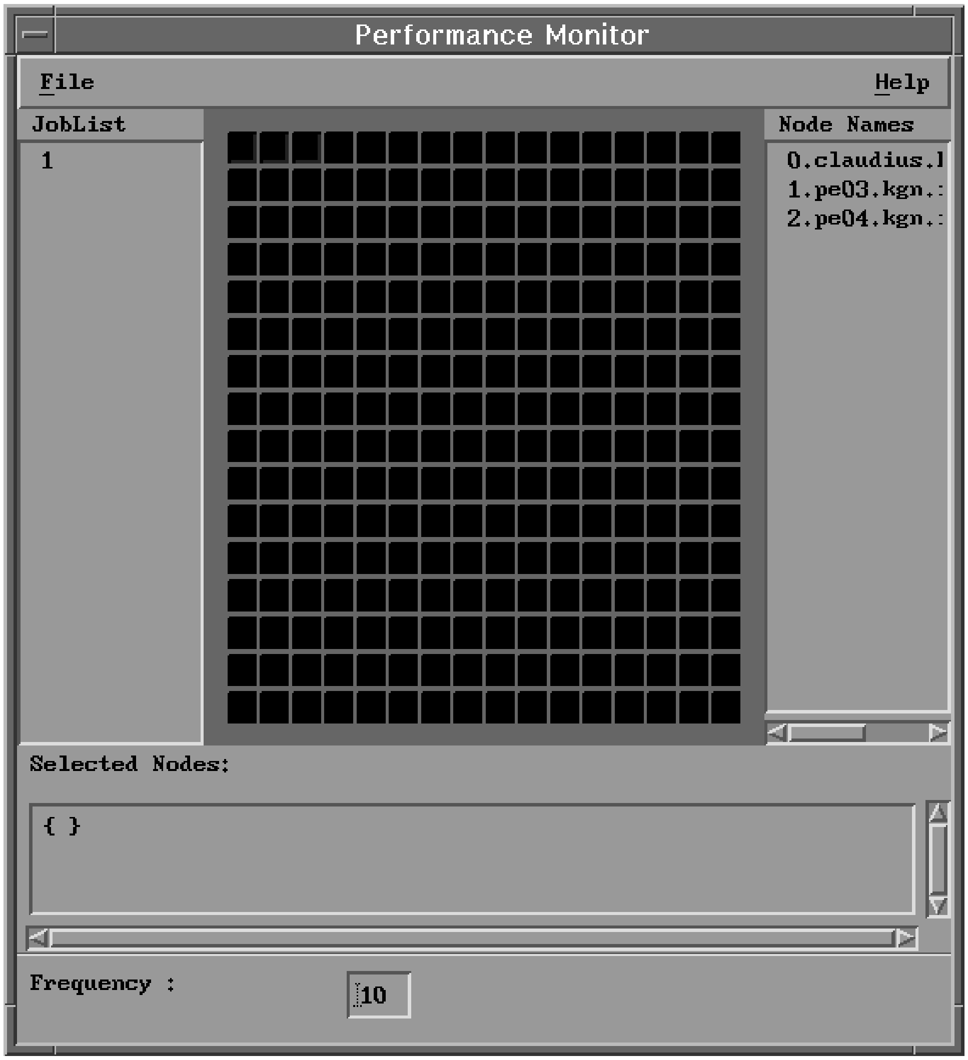


Figure 31. Performance Monitor Window

This window consists of:

- A *Job List*. This lists all the jobs currently running on the SP 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 an RS/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 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 143.
Start monitoring the selected processor nodes.	"Step 2: Start Monitoring" on page 143.
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 144.

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 114. 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 an SP 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 an RS/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 114. 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 at the specified interval 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.

SELECT **File → Monitor**

- Sampling is restarted and VT advances the views according to the new interval setting.

Adjusting a View's Time Resolution 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 *time resolution*. The actual appearance of the view – the colors it uses – is determined by its *display spectrum*. You can change the time resolution and display spectrum for each view.

Note: When running the Visualization Tool on an X-station, you may find that the color definitions will vary from that of an RS/6000 display. X-stations do not always have access to all possible colors. Refer to the documentation supplied with your X-station for more information on color definitions.

To adjust a view's time resolution and select the display spectrum:

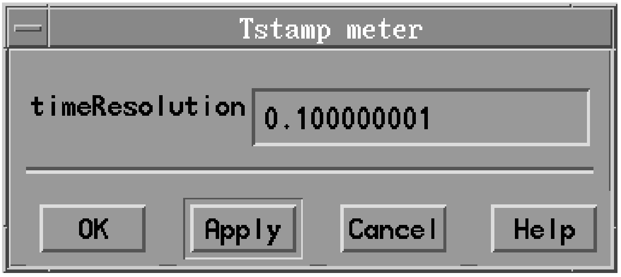
PLACE the cursor over the view.

PRESS the right mouse button.

- A selection menu appears.

SELECT **configuration**

- The view's Time Resolution window opens.



Tstamp meter

timeResolution 0.100000001

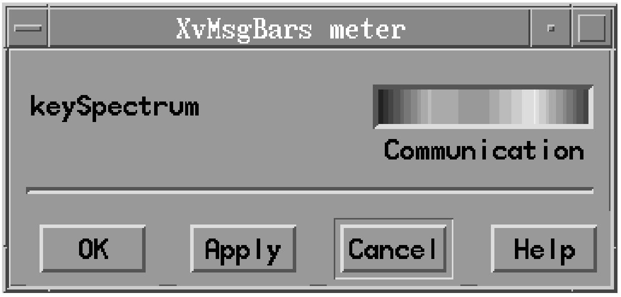
Figure 32. Time Resolution Window

FOCUS on the Time Resolution input field.

TYPE IN the new time resolution 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 148.



XvMsgBars meter

keySpectrum Communication

Figure 33. 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. See the next section, Adjusting View Colors, for advice on choosing the appropriate spectrum for this view.

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 time resolution and color. “View Descriptions” on page 148 describes, for each of the views, any additional parameters you can adjust.

Adjusting View Colors

VT allocates seven different spectrums for use by the different views. Some of the initial colors in each spectrum are determined by a set of X resources which can be found in the *Xdefaults* file, or in Appendix E, “Customizing Tool Resources” on page 247. The seven different spectrums are

1. CPU Load Spectrum

   This spectrum is intended for use by the System Summary view. There is one color allocated to represent each of the possible kernel states: idle, user, kernel, wait and other.

2. Communications Spectrum

   This spectrum is intended for use by the Interprocessor Communication view. By setting appropriate resources, a unique color can be assigned for each possible communication event. However the number of unique events is quite large and allocating a unique color for each will most likely exceed the set of available colors. Additional resources are available which will assign a single color to all events in a particular group. For example, you can give all MPL events one color and all MPI events another color. If you don't need to worry about what MPL is doing, all its events can be colored black.

   You can also assign one color to each group of message passing events. You can make point-to-point events one color and CCL events another color. Going one step further, you can make your send events a different color from your receive events, and color your blocking events differently from your nonblocking events. The default resource file shipped with VT specifies colors for these groups instead of colors for each unique event type.

   For individual events, you can still assign separate colors by specifying the resource name of the event for which you want to define the color.

3. Random Spectrum

   This spectrum is intended for use by the System views. There are 20 different colors allocated for identifying unique events or states in the views.

   The allocation is done internally by varying the color intensity (hue, saturation and brightness) in each pass of a loop. Hue is initially 60 degrees (yellow). Each successive hue is 115 degrees away from the previous hue (modulo 360). Brightness will initially be 1.0 and will alternate between 1.0 and 0.75. Initially saturation will be 1.0 for the first 2 colors and 0.5 for the next two colors and then will repeat this pattern. This produces a set of colors where each entry is easily discernible from its neighbor, but may be similar to other entries.

4. Discrete Spectrum

   This spectrum is intended for use by the System views. There are 10 different colors allocated for identifying unique events or states in the views. The allocation is done internally by varying the color intensities across a range starting with a deep blue and ending with an orange. This scheme produces a set of colors such that each entry in the spectrum has enough contrast to make it easily discernible from all other entries.

5. Fade Spectrum
6. Monochrome Spectrum

   These spectrums are intended for use by the Message Status Matrix and Processor Utilization (3D) views. The color intensity is incremented from black to white (monochrome spectrum) or black to a user specified color (fade spectrum). The intensity of an event serves as index into the spectrum such that extreme events (such as a spike in processor utilization) will be easily discernible from less intense events.
7. Continuous Spectrum

   Like the Monochrome and Fade spectrums, this spectrum is intended for use where the value associated with an event is reflected by the intensity of its color. 100 colors are allocated for this spectrum, with a user definable start and end color. The default start color is (cool) blue and the end is (hot) red.

VT could use 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*.

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 suggested 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

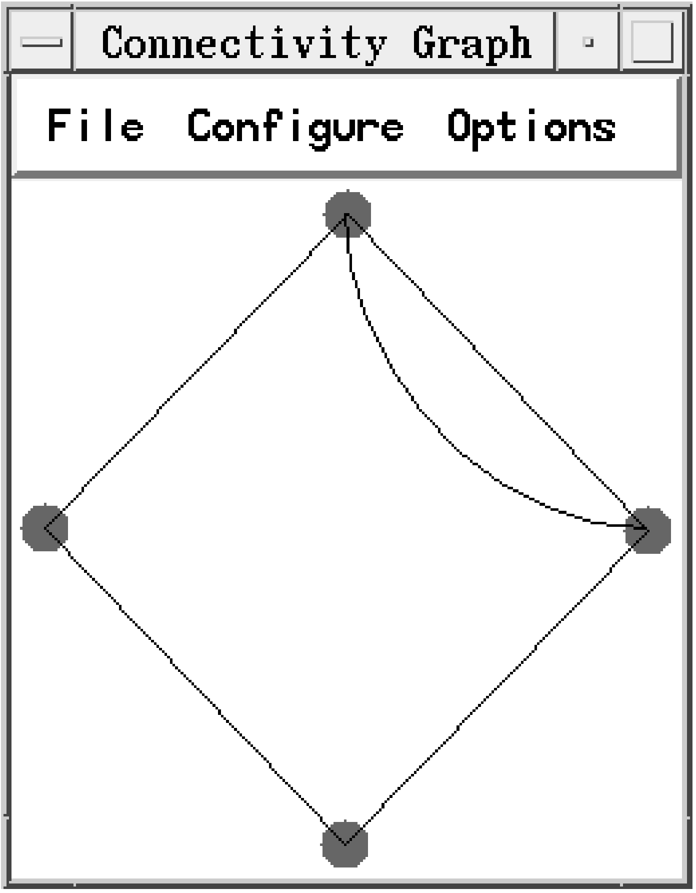
For more information on the trace records included in each type, refer to “Types of Trace Records” on page 117.

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. The color of the connected nodes depends on what particular communication call is being displayed. This is an instantaneous view.

Notes:

1. Some VT dialogs, such as the Connectivity Graph, include a menu bar at the top of the window showing *File*, *Config*, and *Options*. This menu bar is similar to the menu that appears after pressing the third mouse button of a three-button mouse, which provides options for rearranging the display.
2. If the Connectivity Graph and message bar displays use the same spectrum, the color of the Connectivity Graph nodes will match the color of the process labels on the Interprocessor Communication display.
3. A dangled arc may appear on the Connectivity Graph if the trace records timestamp is not synchronized properly. See "Trace Record Timestamps" on page 118 for more information.



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:

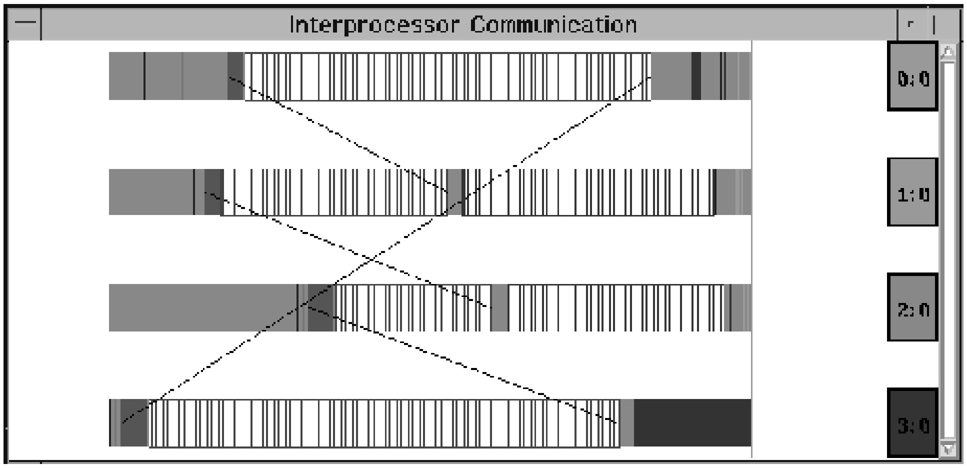
SELECT **Options → Cumulative**

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 the communication thread number within that node, 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, nonblocking 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 between all participants in 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 nonblocking sends to the ends of blocking receives, waits and status events. Lines are only drawn when the wait or status for any nonblocking calls involved in the communication have completed. This is a streaming view.

Note: Generally tracing can be turned off with no impact to the matching events in interprocessor communication.



Using the Search Feature: This view has a search feature that helps you to easily locate the communication events and processor nodes you are interested in.

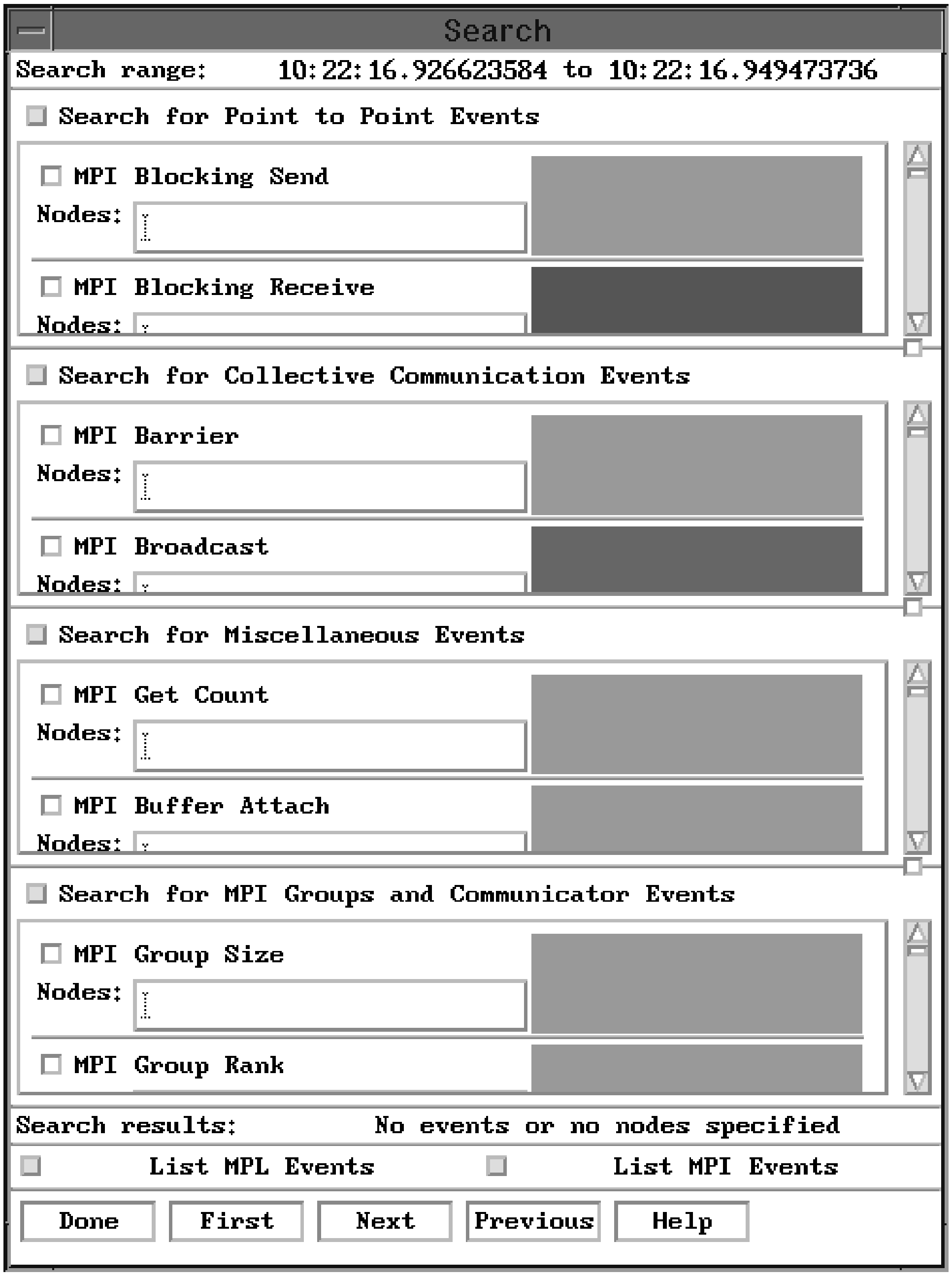
PLACE the mouse cursor over the view.

PRESS the right mouse button.

- A selection menu appears.

SELECT **Search**

- The Search window opens. You are now in search mode.



Search

Search range: 10:22:16.926623584 to 10:22:16.949473736

□ Search for Point to Point Events

□ MPI Blocking Send

Nodes:

□ MPI Blocking Receive

Nodes:

□ Search for Collective Communication Events

□ MPI Barrier

Nodes:

□ MPI Broadcast

Nodes:

□ Search for Miscellaneous Events

□ MPI Get Count

Nodes:

□ MPI Buffer Attach

Nodes:

□ Search for MPI Groups and Communicator Events

□ MPI Group Size

Nodes:

□ MPI Group Rank

Search results: No events or no nodes specified

□ List MPL Events □ List MPI Events

Figure 34. The Search window

The Search window is arranged as follows:

- Search range - shows the time of the first and last event in the display's history buffer. The Search function is limited to the range of events currently stored in the buffer.
- Search for...Events - allows you to search for specific categories of events:
  - Point-to-Point
  - Collective Communication
  - Miscellaneous
  - MPI Groups and Communicator.

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.

Note: In the text field you specify the nodes to be searched in the following way:

- Individually: `1 2` or `1,2`
- By range: `1 - 3`
- Selecting all: `-`

- Search results - depends on the information specified. You will see either

  `No events or no nodes specified`

  or the following

  `Searched to` *`beginning/end`* `(of range)`

  `Event found at` *`time`*

- List MPL/MPI Events - allows you to save space by only listing events of interest.
- Search options

There are four buttons for locating occurrences of the specified pattern: **First**, **Next**, **Previous**, and **Last**.

The following is an example of how to use the search feature:

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 see the last blocking send involving the two nodes:

PRESS **Last**

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**, **Previous**, or **Last** shows you the first, next, previous or last 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.

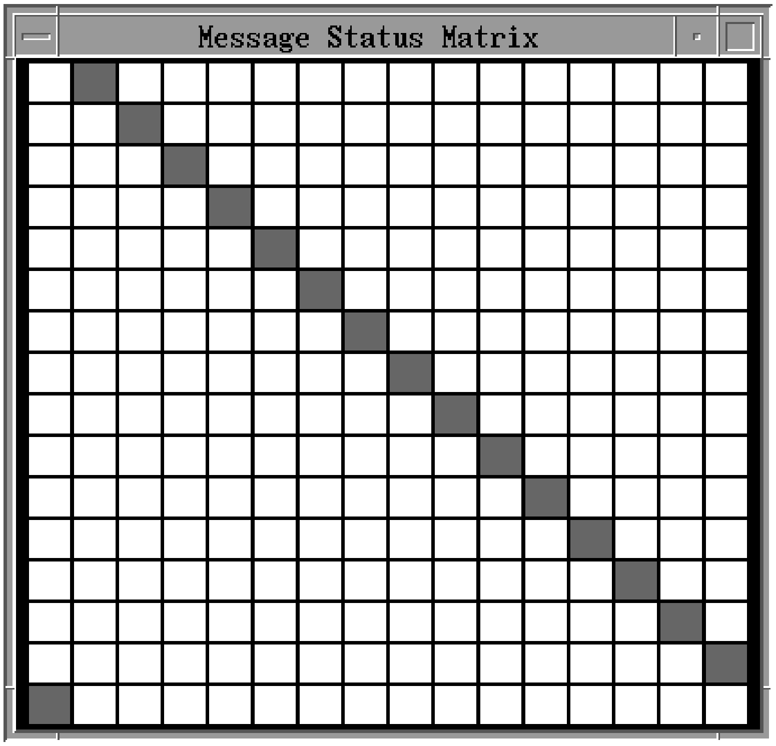
- A window opens displaying the:
  - time index. This is the time during the program's execution that is currently being depicted in the view.
  - node number
  - the type of communication event
  - the time the event occurred.

  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 Time Resolution and Colors” on page 144, 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 Appendix E, “Customizing Tool Resources” on page 247.

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. The color of the rectangles indicates the size of the instantaneous or cumulative message. Smaller messages are colored from the left-hand side of the spectrum; larger messages, from the right-hand side. 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 this display is started after trace playback has begun, the cumulative information will only be shown with respect to when the display was opened.

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**

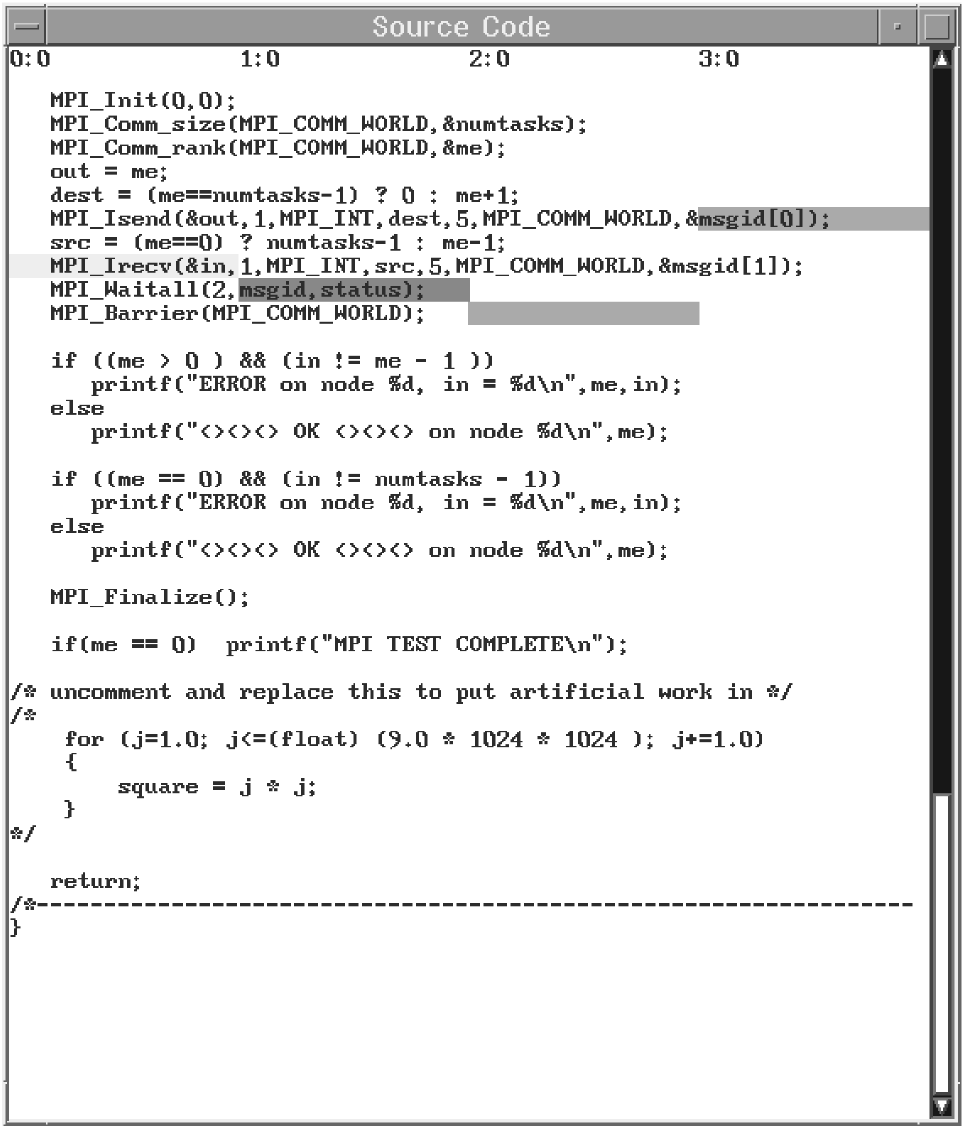
Source Code

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 and the corresponding communication thread numbers. 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.

In order to use this view, you must also have compiled your program with the **-g** flag, and the executable must be accessable from the current $PATH environment variable. 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 159 describes how to add directories to this source code search path. This is an instantaneous view.

The source code view only displays information for one executable, by default, the executable running on task 0. In the SPMD model, all tasks run the same executable, but for MPMD different tasks will run different executables. To make VT display program information for an executable running on a task other than task 0, you must specify that task with the comand line flag **-mp_source**.

For example, consider an MPMD program with executables named **master** and **worker**. **Master** runs on task 0 and **worker** runs on tasks 1 and 2. By default, the source code view will display the source code for **master** and track the execution of task 0 through that code. To track the execution of **worker** through its source, you would start VT with **vt -mp_source 1**.



```
   MPI_Init(0,0);
   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&me);
   out = me;
   dest = (me==numtasks-1) ? 0 : me+1;
   MPI_Isend(&out,1,MPI_INT,dest,5,MPI_COMM_WORLD,&msgid[0]);
   src = (me==0) ? numtasks-1 : me-1;
   MPI_Irecv(&in,1,MPI_INT,src,5,MPI_COMM_WORLD,&msgid[1]);
   MPI_Waitall(2,msgid,status);
   MPI_Barrier(MPI_COMM_WORLD);

   if ((me > 0 ) && (in != me - 1 ))
      printf("ERROR on node %d, in = %d\n",me,in);
   else
      printf("<><><> OK <><><> on node %d\n",me);

   if ((me == 0) && (in != numtasks - 1))
      printf("ERROR on node %d, in = %d\n",me,in);
   else
      printf("<><><> OK <><><> on node %d\n",me);

   MPI_Finalize();

   if(me == 0)  printf("MPI TEST COMPLETE\n");

/* uncomment and replace this to put artificial work in */
/*
    for (j=1.0; j<=(float) (9.0 * 1024 * 1024 ); j+=1.0)
    {
        square = j * j;
    }
*/

   return;
/*-----------------------------------------------------------------
}
```

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 &ndash 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**

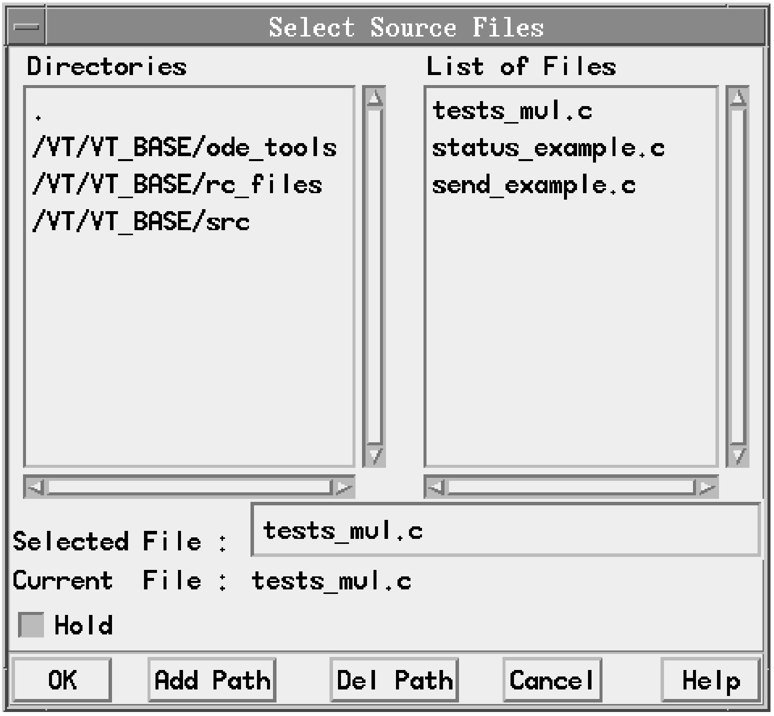
Adding and Deleting Paths to the Source: In order to display a source code file in this view, it must be in your current directory or in a special source code search path. When you start VT, you can indicate a search path to a program's source code using the **-spath** flag. See Table 12 on page 113.

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.



Directories:

.
/VT/VT_BASE/ode_tools
/VT/VT_BASE/rc_files
/VT/VT_BASE/src

List of Files:

tests_mul.c
status_example.c
send_example.c

Selected File : tests_mul.c
Current File : tests_mul.c

Figure 35. 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:

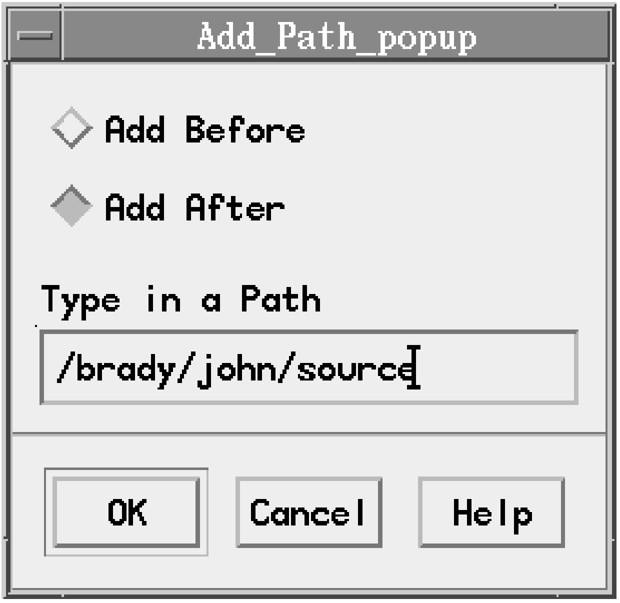
PLACE the mouse cursor over the name of one of the directories listed.

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.



◇ Add Before

◆ Add After

Type in a Path

/brady/john/source

Figure 36. 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 35 on page 160) 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 117.

Kernel Utilization (Bar Chart)

This view uses a bar chart to visualize the amount of time the CPU spent executing the kernel function. 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.

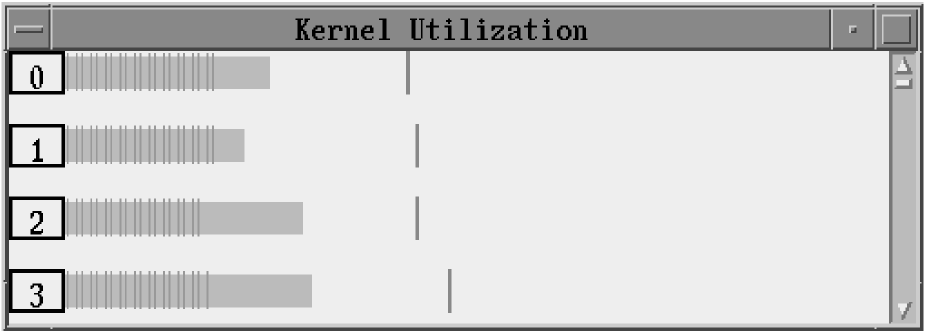


Figure 37. Kernel Utilization (Bar Chart)

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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

Kernel Utilization (Graph)

This view uses a strip graph, or a group of strip graphs, to visualize the amount of time the CPU spent executing the kernel function. 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.

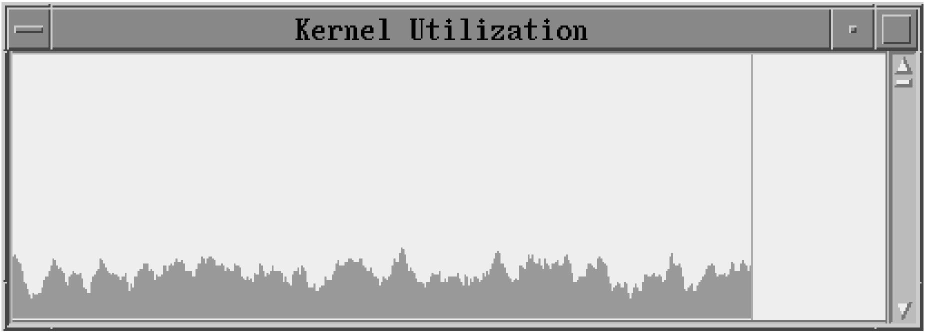


Figure 38. Kernel Utilization (Graph)

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.

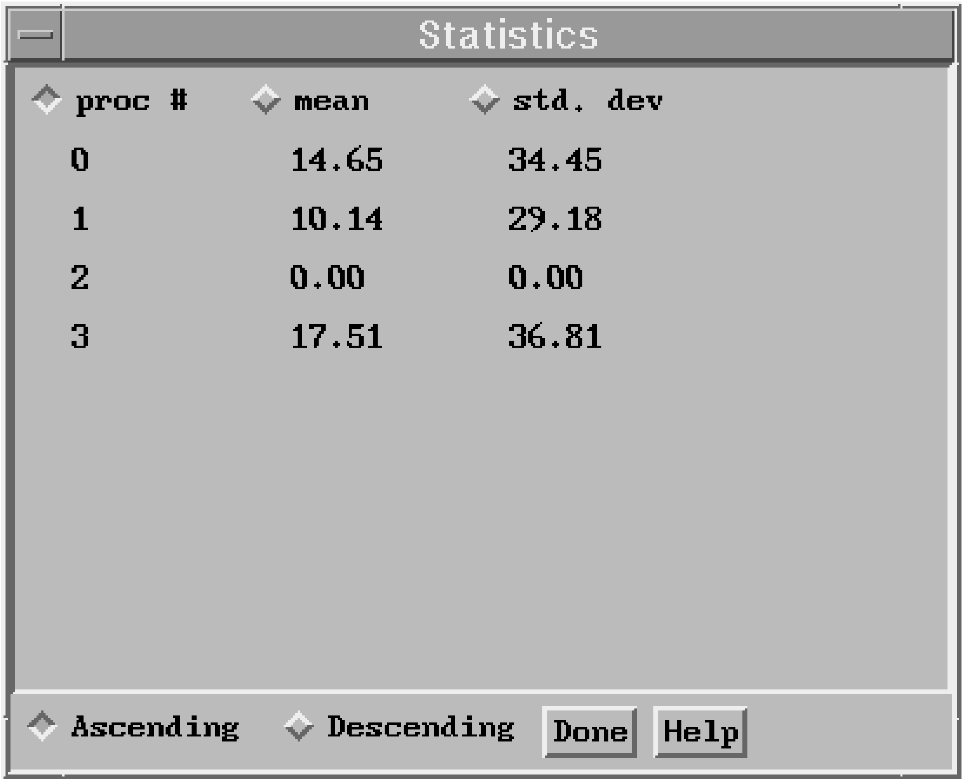
Toggling Between Display Modes: This view has two display modes – aggregate and individual. By default the view is in aggregate display mode – it uses a single strip graph. To display a separate strip graph for each node:

PLACE the mouse cursor over the view.

PRESS the right mouse button.

- A selection menu appears.

SELECT **Individual**



◇ proc #	◇ mean	◇ std. dev
0	14.65	34.45
1	10.14	29.18
2	0.00	0.00
3	17.51	36.81

Figure 39. Statistics window.

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, as shown in Figure 39.

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 in Figure 37 on page 163. 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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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 165.

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 in Figure 37 on page 163. 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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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 165.

User Utilization (Bar Chart)

This view uses a bar chart to visualize the amount of time the CPU spent executing the user code. 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 in Figure 37 on page 163. 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 The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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.

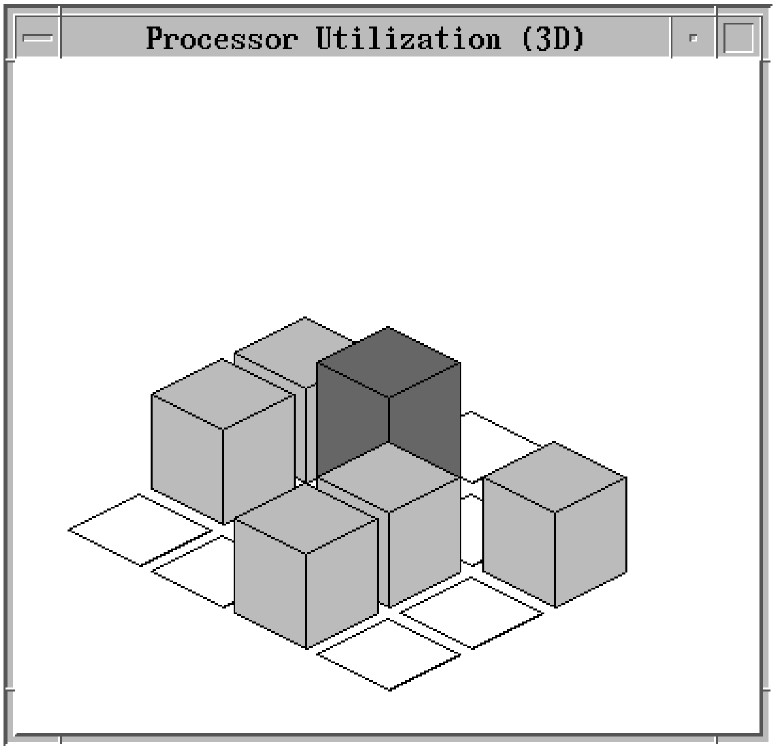
● A selection menu appears.

SELECT **Show Statistics**

● The Statistics Window opens. This window is shown on page 165.

Processor Utilization (3D Bar Chart)

This view is similar to the user utilization bar chart. It visualizes the amount of time the CPU spent executing the kernel code, as well as the user code. In this view, 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**

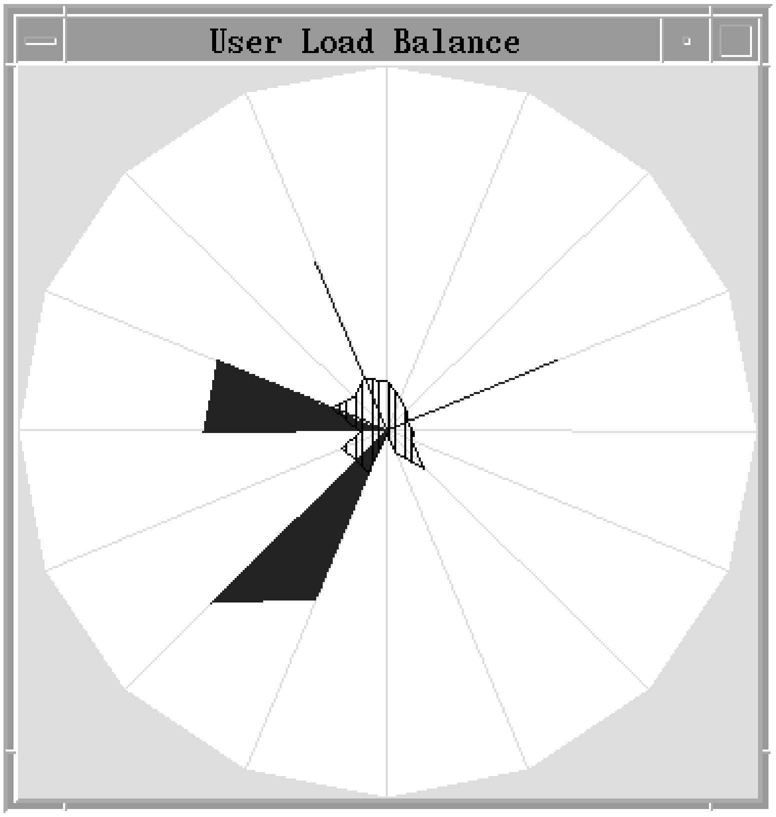
- The view's Parameters window opens.

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 for that node, 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 117.

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 in Figure 37 on page 163. 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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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 165.

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 in Figure 37 on page 163. 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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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 165.

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 in Figure 37 on page 163. 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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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 165.

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 117.

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 in Figure 37 on page 163. 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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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 165.

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 in Figure 37 on page 163. 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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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 165.

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 117.

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 in Figure 37 on page 163. 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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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 165.

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 in Figure 37 on page 163. 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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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 165.

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 in Figure 37 on page 163. 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. The purpose of the Recalibrate button on the configuration pop-up window is to enable or disable the automatic recalibrate function.

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 in Figure 38 on page 164. 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**

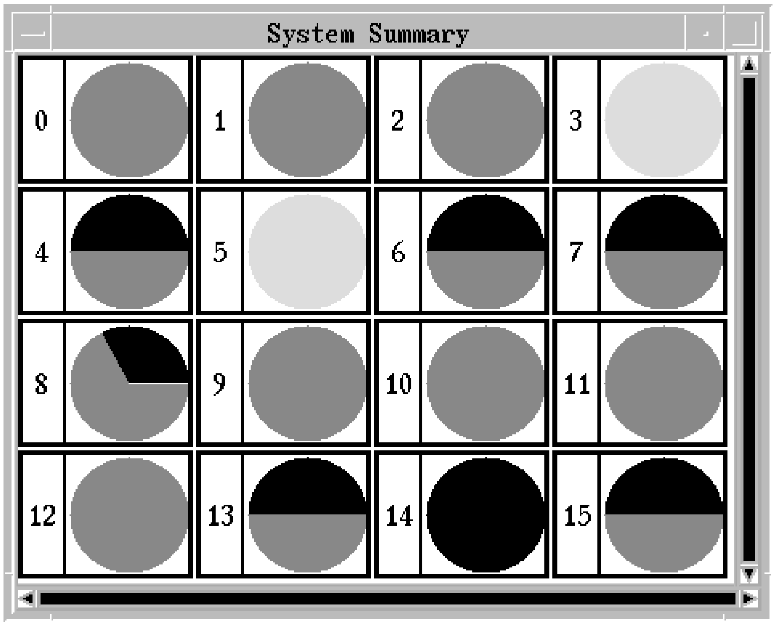
● The Statistics Window opens. This window is shown on page 165.

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 Time Resolution and Colors” on page 144, 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.

Appendix A. Parallel Environment Tools Commands

This appendix contains the manual pages for the PE tools commands discussed throughout this book. Each manual page is organized into the sections listed below. The sections always appear in the same order, but some appear in all manual pages while others are optional.

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 "Typographic Conventions" on page xii.

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.

pdbx

NAME

pdbx – Invokes the **pdbx** debugger, which is the command-line debugger built on **dbx**.

SYNOPSIS

pdbx [*program* [*program_options*]] [*poe options*]
[**-c** *command_file*]
[**-d** *nesting_depth*]
[**-I** *directory*
[**-I** *directory*]...]
[**-F**]
[**-x**]

pdbx -a *poe process id*
[*limited poe options*]
[**-c** *command_file*]
[**-d** *nesting_depth*]

[**-I** *directory*
[**-I** *directory*]...]
[**-F**]
[**-x**]

pdbx -h

The **pdbx** command invokes the **pdbx** debugger. This tool is based on the **dbx** debugger, but adds function specific to parallel programming.

FLAGS

Because **pdbx** runs in the Parallel Operating Environment, it accepts all the flags supported by the **poe** command.

Note: **poe** uses the **PATH** environment variable to find the program, while **pdbx** does not.

See the **poe** manual page in *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for a description of these options. Additional **pdbx** flags are:

-a Attaches to a running **poe** job by specifying its process id. This must be executed from the node where the **poe** job was initiated. When using the debugger in attach mode there are some debugger command line arguments that should not be used. In general, any arguments that control how the partition is set up or specify application names and arguments should not be used.

-c Reads startup commands from the specified *commands_file*.

-d Sets the limit for the nesting of program blocks. The default nesting depth limit is 25.

-F This flag can be used to turn off *lazy reading* mode. Turning lazy reading mode off forces the remote **dbx** sessions to read all symbol table information at startup time. By default, lazy reading mode is on.

Lazy reading mode is useful when debugging large executable files, or when paging space is low. With lazy reading mode on, only the required symbol table information is read upon initialization of the remote **dbx** sessions. Because all symbol table information is not read at **dbx** startup time when in lazy reading mode, local variable and related type information will not be initially available for functions defined in other files. The effect of this can be seen with the **whereis** command, where instances of the specified local variable may not be found until the other files containing these instances are somehow referenced.

-h Writes the **pdbx** usage to STDERR then exits. This includes **pdbx** command line syntax and a description of **pdbx** options.

-I (upper-case i)
Specifies a *directory* to be searched for an executable's source files. This flag must be specified multiple times to set multiple

paths. (Once **pdbx** is running, this list can be overridden on a group or single node basis with the **use** subcommand.)

-x Prevents **dbx** 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. For more information on .pdbxinit see Table 2 on page 5 and “Reading Subcommands From a Command File” on page 36. This is frequently local, but 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

pdbx is the Parallel Environment's command-line debugger for parallel programs. It is based, and built, on the AIX debugging tool **dbx**.

pdbx 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. It is also advisable to not use the optimization option, **-O**. Using the debugger on optimized code may produce inconsistent and erroneous results. For more information on the **-g** and **-O** compiler options, refer to their use on other compiler commands such as **cc** and **xlf**. These compiler commands are described in *IBM AIX Version 4 Commands Reference*

pdbx maintains **dbx’s** command-line interface and subcommands. When you invoke **pdbx**, the **pdbx** command prompt displays to mark the start of a **pdbx** session.

When using **pdbx**, 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 you define. Context sensitive subcommands, when entered, only affect those tasks in the current command context. Context insensitive subcommands are not affected by the command context setting.

If you are already familiar with **dbx**, you should be aware that some **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**, **screen**, and the **sh** subcommand with no arguments.
- 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**.

When a task hangs (there is no **pdbx** prompt) you can press <**Ctrl-c**> to acquire control. This displays the **pdbx** subset prompt `pdbx-subset([group | task])`, and provides a subset of **pdbx** functionality:

- Changing the current context
- Displaying information about groups/tasks
- Interrupting the application
- Showing breakpoint/tracepoint status
- Getting help
- Exiting the debugger.

You can change the subset of tasks to which context sensitive commands are directed. Also, you can understand more about the current state of the application, and gain control of your application at any time, not just at user-defined breakpoints.

At the **pdbx** subset prompt, all input you type at the command line is intercepted by **pdbx**. All commands are interpreted and operated on by the home node. No data is passed to the remote nodes and STDIN is not given to the application. Most commands at the **pdbx** subset prompt produce information about the application and then produce another **pdbx** subset prompt. The exceptions are the **halt**, **back**, **on**, and **quit** commands. For more information, see “Context Switch when Blocked” on page 17.

ENVIRONMENT VARIABLES

Because the **pdbx** command runs in the Parallel Operating Environment, it interacts with the same environment variables associated with the **poe** command. See the **poe** manual page in *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for a description of these environment variables. As indicated by the syntax statements, you are also able to specify **poe** command line options when invoking **pdbx**. Using these options will override the setting of the corresponding environment variable, as is the case when invoking a parallel program with the **poe** command. Additional variables are:

HOME
During **pdbx** initialization, **pdbx** uses this environment variable to search for two special initialization files. First, **pdbx** searches for *.pdbxinit* in the user's current directory. If the file is not found, **pdbx** checks the file *$HOME/.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 needed in rare situations 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, but may be a shared directory. If not set, and if its associated command-line flag **tmpdir** is not used, the default location is */tmp*.

MP_DEBUG_INITIAL_STOP

This environment variable redefines the initial stop point in **pdbx** (overriding the stop in main). It can be set to *sourcefile:linenumber*, where *sourcefile* is a file containing source code of the program to be executed. Typically, the source file name ends with the **.c**, **.C**, or **f** suffix. *Linenumber* is a line number in this file. This line must contain executable code, not data declarations or Fortran FORMAT statements. It cannot be a comment, blank, or continuation line.

If no *linenumber* is specified (and the colon is omitted), the *sourcefile* field is taken to be a function or subroutine name, and a "stop in" is performed on entry to the function.

If **MP_DEBUG_INITIAL_STOP** is undefined, the default stop location will be the first executable line in the function main. For Fortran source programs, it will be the first executable line in the main program.

EXAMPLES

To start **pdbx**, first set up the execution environment as you would for the **poe** command, and then enter:

```
pdbx
```

After initialization, you should see the prompt:

```
pdbx(all)
```

FILES

.pdbxinit (Initial commands for **pdbx** in *./* or *$HOME*)

.pdbxinit.*process_id.task_id* (Initial commands for the individual **dbx** tasks)

For more information on .pdbxinit see Table 2 on page 5 and "Reading Subcommands From a Command File" on page 36.

Note: The following temporary files are created during the execution of **pdbx** in attach mode:

- */tmp/.pdbx.<poe-pid>.host.list* - a temporary host list file containing information needed to attach to tasks on remote nodes.
- */tmp/.pdbx.<pdbx-pid>.menu* - a temporary file to hold the attach task menu. Both of these files are removed before the debugger exits.

RELATED INFORMATION

Commands: **dbx**(1), **mpcc**(1), **mpcc_r**(1), **mpCC**(1), **mpCC_r**(1), **mpxlf**(1), **mpxlf_r**(1), **pedb**(1), **poe**(1)

pdbx SUBCOMMANDS

pdbx alias Subcommand

alias [*alias_name* [*alias_string*]]

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
th	thread
mu	mutex
cv	condition
attr	attribute

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.

1. If you have two task groups defined in your **pdbx** session called “master” and “workers”, and you wish to define aliases to easily qualify each, enter:

   ```
   alias mas on master

   alias w on workers
   ```

   This will allow you to switch the command context between the master and workers groups by typing:

   ```
   mas
   ```

   to switch context to the “master” group, or:

   ```
   w
   ```

   to switch context to the “workers” 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 to this subcommand is the **pdbx unalias** subcommand.

pdbx assign Subcommand

assign <*variable*> = <*expression*>

The **assign** subcommand assigns the value of an expression to a variable.

1. To assign a value of 5 to the x variable:

   ```
   pdbx(all) assign x = 5
   ```

2. To assign the value of the y variable to the x variable:

   ```
   pdbx(all) assign x = y
   ```

3. To assign the character value ‘z’ to the z variable:

   ```
   pdbx(all) assign z = 'z'
   ```

4. To assign the boolean value false to the logical type variable B:

   ```
   pdbx(all) assign B = false
   ```

5. To assign the “Hello World” string to a character pointer Y:

   ```
   pdbx(all) assign Y = "Hello World"
   ```

6. To disable type checking, activate the set variable $unsafeassign:

   ```
   pdbx(all) set $unsafeassign
   ```

pdbx attach Subcommand

attach all

attach <*task_list*>

The **attach** subcommand is used to attach the debugger to some or all the tasks of a given **poe** job.

Individual tasks are separated by spaces. A range of tasks may be separated by a dash or a colon. For example, the command **attach 2 4 5-7** would mean to attach to tasks 2,4,5,6, and 7.

pdbx attribute Subcommand

attribute
attribute [<*attribute_number*> ...]

The **attribute** subcommand displays information about the user thread, mutex, or condition attributes objects defined by the *attribute_number* parameters. If no parameters are specified, all attributes objects are listed.

For each attributes object listed, the following information is displayed:

attr Indicates the symbolic name of the attributes object, in the form $a*attribute_number*.

obj_addr Indicates the address of the attributes object.

type Indicates the type of the attributes object; this can be **thr**, **mutex**, or **cond** for user threads, mutexes, and condition variables respectively.

state Indicates the state of the attributes object. This can be valid or invalid.

stack Indicates the stacksize attribute of a thread attributes object.

scope Indicates the scope attribute of a thread attributes object. This determines the contention scope of the thread, and defines the set of threads with which it must contend for processing resources. The value can be sys or pro for system or process contention scope.

prio Indicates the priority attribute of a thread attributes object.

sched Indicates the **schedpolicy** attribute of a thread attributes object. This attribute controls scheduling policy, and can be fifo (first in first out), rr (round robin), or other.

p-shar Indicates the process-shared attribute of a mutex or condition attribute object. A mutex or condition is process-shared if it can be accessed by threads belonging to different processes. The value can be yes or no.

protocol Indicates the protocol attribute of a mutex. This attribute determines the effect of holding the mutex on a thread's priority. The value can be no_prio, prio, or protect.

Related to this subcommand are the **condition mutex** and **thread** subcommands.

pdbx back Subcommand

back

The **back** command returns you to a **pdbx** prompt when you were already at a **pdbx** subset prompt. You can use the command if you want the application to continue as it was before <**Ctrl-c**> was issued. Also, you can use it at the **pdbx** subset prompt if all of the nodes are checked into “debug ready” state, and you want to do full **pdbx** processing.

The **back** command is only valid at the **pdbx** subset prompt.

pdbx call Subcommand

call <*procedure*> (<*parameters*>)

The **call** subcommand runs a procedure specified by the procedure parameter. The return code is not printed. If any parameters are specified, they are passed to the procedure being run.

The program stack will be returned to its previous state after the procedure specified by **call** completes. Any side effect of the procedure, such as global variable updates, will remain.

Related to this subcommand is the **print** subcommand.

pdbx case Subcommand

case [**default** | **mixed** | **lower** | **upper**]

The **case** subcommand changes how **pdbx** interprets symbols. The default handling of symbols is based on the current language. If the current language is C, C++, or undefined, the symbols are not folded. If the current language is Fortran, the symbols are folded to lowercase. Use this command if a symbol needs to be interpreted in a way not consistent with the current language.

Entering the **case** subcommand with no parameters displays the current case mode. The parameters include:

default Varies with the current language.

mixed Causes symbols to be interpreted as they actually appear.

lower Causes symbols to be interpreted as lowercase.

upper Causes symbols to be interpreted as uppercase.

pdbx catch Subcommand

catch

catch <*signal_number*>

catch <*signal_name*>

The **catch** subcommand with no arguments prints all signals currently being caught. If a signal is specified, **pdbx** will trap the signal before it is sent to the program. This is useful when the program being debugged has signal handlers.

When the program encounters a signal that is being caught to the debugger, a message stating which signal was detected is shown, and the **pdbx** prompt is displayed. To have the program continue and process the signal, issue the **cont** subcommand with the **signal** option. Other execution control commands and the **cont** subcommand without the **signal** option will cause the program to behave as if it had never encountered the signal.

A signal may be specified by number or name. Signal names are by default case insensitive and the "SIG" prefix is optional.

By default all signals are caught except SIGHUP, SIGKILL, SIGPIPE, SIGALRM, SIGCHLD, SIGIO and SIGVIRT. When debugging a threaded application (including those compiled with **mpcc_r**, **mpCC_r** or **mpxlf_r**), all signals are caught except SIGHUP, SIGKILL, SIGALRM, SIGCHLD, SIGIO and SIGVIRT.

Related to this subcommand are the **ignore** and **cont** subcommands.

pdbx condition Subcommand

condition
condition [<*condition_number*> ...]
condition [**wait** | **nowait**]

The **condition** subcommand displays the current state of all known conditions in the process. Condition variables to be listed can be specified through the <*condition_number*> parameters, or all condition variables will be listed. Users can also choose to display only condition variables with or without waiters by using the **wait** or **nowait** options.

The information listed for each condition is as follows:

cv Indicates the symbolic name of the condition variable, in the form $c*condition_number*.

obj_addr Indicates the memory address of the condition variable.

num_wait Indicates the number of threads waiting on the condition variable.

waiters Lists the user threads which are waiting on the condition variable.

Related to this subcommand are the **attribute mutex** and **thread** subcommands.

pdbx cont Subcommand

cont

cont <*signal_number*>

cont <*signal_name*>

The **cont** subcommand allows execution to continue from where the program last stopped, until either the program finishes or another breakpoint is reached. If a signal is specified, it is given to the program, and the process continues as though it received the signal. If a signal is not specified, the process continues as though it had not been stopped.

Related to this subcommand are the **catch**, **ignore**, **step**, **stepi**, **next**, and **nexti** subcommands.

pdbx dbx Subcommand

dbx *dbx_subcommand*

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**, **screen**, and the **sh** subcommand with no arguments are currently unsupported under **pdbx** and should not be used.

To display the events that the **dbx** running as task 1 recognizes, enter:

```
on 1 dbx status
```

Related to this subcommand is the **dbx** command.

pdbx delete Subcommand

delete [*event_list*] **|** [*] **|** [**all**]

The **delete** subcommand removes events (breakpoints and tracepoints) of the specified event numbers. An event list can be specified in the following manner. To indicate a range of events, enter the first and last event numbers, separated by a colon or dash. To indicate individual events, enter the numbers, separated by a space or comma. You can specify “ * ”, which deletes all events that were created in the current context. You can also specify “all”, which deletes all events, regardless of context.

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

The output of the status command shows the context from which the event was created. Event numbers 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.

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, you would do one of the following:

```
on 1

delete 0

on all

delete 0,1


OR


on 1

delete 0

on all

delete *


OR


delete all
```

Related to this subcommand are the **pdbx status**, **stop**, and **trace** subcommands.

pdbx detach Subcommand

detach

The **detach** subcommand detaches **pdbx** from all tasks that were attached. This subcommand causes the debugger to exit but leaves the **poe** application running.

pdbx dhelp Subcommand

dhelp

dhelp <*dbx_command*>

The **dhelp** command with no arguments displays a list of **dbx** commands about which detailed information is available.

If you type **dhelp** with an argument, information will be displayed about that command.

Note: The partition must be loaded before you can use this command, because it invokes the **dbx help** command. It is also required that a task be in “debug ready” state to process this command.

Related to this subcommand is the **pdbx help** subcommand.

pdbx display memory Subcommand

<*address*> / [<*mode*>]
<*address*> , <*address*> / [<*mode*>]
<*address*> / [<*count*>] [<*mode*>]

The **display memory** subcommand, which does not have a keyword to initiate the command, displays a portion of memory controlled by the address(es), count(s) and mode(s) specified.

If an address is specified, the display contents of memory at that address is printed. If more than one address or count locations are specified, display contents of memory starting at the first <*address*> up to the second <*address*> or until <*count*> items are printed. If the address is “.,” the address following the one most recently printed is used. The mode specifies how memory is to be printed. If it is omitted the previous mode specified is used. The initial mode is “X.”

The following modes are supported:

i	print the machine instruction
d	print a short word in decimal
D	print a long word in decimal
o	print a short word in octal
O	print a long word in octal
x	print a short word in hexadecimal
X	print a long word in hexadecimal
b	print a byte in octal
c	print a byte as a character
h	print a byte in hexadecimal
s	print a string (terminated by a null byte)
f	print a single precision real number
g	print a double precision real number
q	print a quad precision real number
lld	print an 8 byte signed decimal number
llu	print an 8 byte unsigned decimal number
llx	print an 8 byte unsigned hexadecimal number
llo	print an 8 byte unsigned octal number

pdbx down Subcommand

down [*count*]

The **down** subcommand moves the current function down the stack the number of levels specified by *count*. The current function is used for resolving names. The default for the *count* parameter is one.

The **up** and **down** subcommands can be used to navigate through the call stack. Using these subcommands to change the current function also causes the current file and local variables to be updated to the chosen stack level.

Related to this subcommand are the **up**, **print**, **dump**, **func**, **file**, and **where** commands.

pdbx dump Subcommand

dump

dump <*procedure*>

dump .

dump <*module name*>

The **dump** subcommand prints the names and values of variables in a given procedure, or the current one if nothing is specified. If the procedure given is ".", then all active variables are printed. If a module name is given, all variables in the module are printed.

Related to this subcommand are the **up**, **down**, **print**, and **where** subcommands.

pdbx file Subcommand

file [*file*]

The **file** subcommand changes the current source file to the file specified by the *file* parameter. It does not write to that file. The *file* parameter can specify a full path name to the file. If the parameter does not specify a path, the **pdbx** program tries to find the file by searching the use path. If the parameter is not specified, the **file** subcommand displays the name of the current source file. The **file** subcommand also displays the full or relative path name of the file if the path is known.

Related to this subcommand is the **func** subcommand.

pdbx func Subcommand

func [*procedure*]

The **func** command changes the current function to the procedure or function specified by the *procedure* parameter. If the *procedure* parameter is not specified, the default current function is displayed. Changing the current function implicitly changes the current source file to the file containing the new function. The current scope used for name resolution is also changed.

Related to this subcommand is the **file** subcommand.

pdbx goto Subcommand

goto <*line_number*>
goto "<*filename*>" : <*line_number*>

The **goto** subcommand causes the specified source line to be run next. Normally, the source line must be in the same function as the current source line. To override this restriction, use the **set** subcommand with the **$unsafegoto** flag.

pdbx gotoi Subcommand

gotoi *address*

The **gotoi** subcommand changes the program counter address to the address specified by the *address* parameter.

pdbx group Subcommand

group add *group_name task_list*

group delete *group_name* [task_list]

group change *old_group_name new_group_name*

group list [*group_name*]

The **group** subcommand groups individual tasks under a common name for easier setting of command context. 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.

Provide a group name that is no longer than 32 characters which starts with an alphabetic character, and is followed by any alphanumeric character combination.

To indicate a range of tasks, enter the first and last task numbers, separated by a colon or dash. To indicate individual tasks, enter the numbers, separated by a space or comma. Individual task identifiers and ranges can also be combined in creating the desired *task_list*.

Note: Group names “all,” “none,” and “attached” are reserved group names. They are used by the debugger and cannot be used in the **group add** or **group delete** commands. However, the group “all” or “attached” can be renamed using the **group change** command, if it currently exists in the debugging session.

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.

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 deleted from the new or existing group.

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 will be followed by a one-letter status indicator.

N	Not loaded	the remote task has not yet been loaded with an executable.
S	Starting	the remote task is being loaded with an executable.
D	Debug ready	the remote task is stopped and debug commands can be issued.
R	Running	the remote task is in control and executing the program.
X	Exited	the remote task has completed execution.
U	Unhooked	the remote task is executing without debugger intervention.
E	Error	the remote task is in an unknown state.

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 was 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 were 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:

   ```
   Group "all" has been renamed to "johnny"
   ```

4. To list all of the groups and the tasks they contain, enter:

```
group list
```

The **pdbx** debugger responds with:

```
johnny    0:D    1:D    2:D    3:D    4:D

first     0:D

rest      1:D    2:D    3:D    4:D
```

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:

   ```
   Task: 1 was successfully deleted from group "rest".

   Task: 2 was successfully deleted from group "rest".

   Task: 3 was successfully deleted from group "rest".
   ```

6. To list all of the groups and the tasks they contain, enter:

   ```
   group list
   ```

   The **pdbx** debugger responds with:

   ```
   allTasks      0:R     1:D     2:D     3:U     4:U     5:D     6:D

                 7:D     8:D     9:D    10:D    11:D

   evenTasks     0:R     2:D     4:U     6:D     8:D    10:R

   oddTasks      1:D     3:U     5:D     7:D     9:D    11:R

   master        0:R

   workers       1:D     2:D     3:U     4:U     5:D     6:D     7:D

                 8:D     9:D    10:R    11:R
   ```

Related to this subcommand is the **pdbx on** subcommand.

pdbx halt Subcommand

halt [**all**]

By using the **halt** command, you interrupt all tasks in the current context that are running. This allows the debugger to gain control of the application at whatever point the running tasks happen to be in the applicaton. To a **dbx** user, this is the same as using <**Ctrl-c**>. This command works at the **pdbx** prompt and **pdbx** subset prompt. If you specify “all” with the command, all running tasks, regardless of context, are interrupted.

Note: At a **pdbx** prompt, the **halt** command never has any effect without “all” specified. This is because by definition, at a **pdbx** prompt, none of the tasks in the current context are in “running” state.

The **halt all** command at the **pdbx** prompt affects tasks outside of the current context. Messages at the prompt show the task numbers that are and are not interrupted, but the **pdbx** prompt returns immediately because the state of the tasks in the current context is unchanged.

When using **halt** at the **pdbx** subset prompt, the **pdbx** prompt occurs when all tasks in the current context have returned to “debug ready” state. If some of the tasks in the current context are running, a message is presented.

Related to this subcommand are the **pdbx tasks** and **group list** subcommands.

pdbx help Subcommand

help - display subjects

help <*subject*> - display details

The **help** command with no arguments displays a list of **pdbx** commands and topics about which detailed information is available.

If you type **help** with one of the **help** commands or topics as the argument, information will be displayed about that subject.

Related to this subcommand is the **pdbx dhelp** subcommand

pdbx hook Subcommand

hook

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.

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:D    1:U    2:D    3:D
```

Tasks marked with the letter D next to them are debug ready, hooked tasks. In this case, tasks 0, 2, and 3 are debug ready. Tasks marked with the letter U are unhooked. In this case, task 1 is unhooked.

Related to this subcommand are the **dbx detach** subcommand and the **pdbx unhook** subcommand.

pdbx ignore Subcommand

ignore

ignore <*signal_number*>

ignore <*signal_name*>

The **ignore** subcommand with no arguments prints all signals currently being ignored. If a signal is specified, **pdbx** stops trapping the signal before it is sent to the program.

A signal may be specified by number or name. Signal names are by default case insensitive and the “SIG” prefix is optional.

All signals except SIGHUP, SIGKILL, SIGPIPE, SIGALRM, SIGCHLD, SIGIO, and SIGVIRT are trapped by default. When debugging a threaded application (including those compiled with **mpcc_r**, **mpCC_r**, or **mpxlf_r**), all signals except SIGHUP, SIGKILL, SIGALRM, SIGCHLD, SIGIO, and SIGVIRT are trapped by default.

The **pdbx** debugger cannot ignore the SIGTRAP signal if it comes from a process outside of the program being debugged.

Related to this subcommand is the **catch** subcommand.

pdbx list Subcommand

list [*procedure* **|** *sourceline-expression*[, *sourceline-expression*]]

The **list** subcommand displays a specified number of lines of the source file. The number of lines displayed is specified in one of two ways:

Tip: Use **on** <*task*> **list**, or specify the ordered standard output option.

- By specifying a procedure using the *procedure* parameter.

  In this case, the **list** subcommand displays lines starting a few lines before the beginning of the specified procedure and until the list window is filled.

- By specifying a starting and ending source line number using the *sourceline-expression* parameter.

  The *sourceline-expression* parameter should consist of a valid line number followed by an optional + (plus sign), or − (minus sign), and an integer. In addition, a *sourceline* of $ (dollar sign) can be used to denote the current line number. A *sourceline* of @ (at sign) can be used to denote the next line number to be listed.

  All lines from the first line number specified to the second line number specified, inclusive, are then displayed, provided these lines fit in the list window.

  If the second source line is omitted, 10 lines are printed, beginning with the line number specified in the *sourceline* parameter.

  If the **list** subcommand is used without parameters, the default number of lines is printed, beginning with the current source line. The default is 10.

  To change the number of lines to list by default, set the special debug program variable, *$listwindow*, to the number of lines you want. Initially, *$listwindow* is set to 10.

To list the lines 1 through 10 in the current file, enter:

```
list 1,10
```

To list 10, or *$listwindow*, lines around the main procedure, enter:

```
list main
```

To list 11 lines around the current line, enter:

```
list $-5,$+5
```

To list the next source line to be executed, issue:

```
pdbx(all) list $

  0:    4       char johnny = 'h';

  1:    4       char johnny = 'h';
```

To just show 1 task, since both are at the same source line:

```
pdbx(all) on 0 list $

  0:    4       char johnny = 'h';
```

To create an alias to list just task 0:

```
pdbx(all) alias l0 on 0 list
```

To list line 5:

```
pdbx(all) l0 5

  0:    5       char jessie  = 'd';
```

To list lines around the procedure `sub`:

```
pdbx(all) l0 sub

  0:   21

  0:   22   /* return ptr to sum of parms, calc and sub1 */

  0:   23   int *sub(char *s, int a, int k)

  0:   24   {

  0:   25       int *tmp;

  0:   26       int it = 0;

  0:   27       int i, j;

  0:   28

  0:   29       /* test calc */

  0:   30       i = 1;

  0:   31       j = i*2;
```

To change the next line to be listed to line 25:

```
pdbx(all) move 25
```

To list the next line to be listed minus two:

```
pdbx(all) l0 @-2

  0:   23   int *sub(char *s, int a, int k)
```

Related to this subcommand is the **dbx list** subcommand.

pdbx listi Subcommand

listi [*procedure* **|** **at** *SourceLine* **|**
address [,*address*]]

The **listi** subcommand displays a specified set of instructions from the current program counter, depending on whether you specify procedure, source line, or address.

The **listi** subcommand with the *procedure* parameter lists instructions from the beginning of the specified procedure until the **list** window is filled.

Using the **at** *SourceLine* flag with the **listi** subcommand displays instructions beginning at the specified source line and continuing until the **list** window is filled. The *SourceLine* variable can be specified as an integer, or as a file name string followed by a : (colon) and an integer.

Specifying a beginning and ending address with the **listi** subcommand, using the *address* parameters, displays all instructions between the two addresses.

If the **listi** subcommand is used without flags or parameters, the next **$listwindow** instructions are displayed. To change the current size of the **list** window, use the **set $listwindow**=*Value* command.

pdbx load Subcommand

load *program* [*program_options*]

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. **pdbx** will look for the program in the current directory unless a relative or absolute pathname is specified. The **load** subcommand is context sensitive. All tasks in the partition must 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.

To load the program “mpprob1” on all tasks in the current context, enter:

```
load mpprob1
```

pdbx map Subcommand

map

The **map** subcommand displays characteristics for each loaded portion of the application. This information includes the name, text origin, text length, data origin, and data length for each loaded module.

pdbx mutex Subcommand

mutex

mutex [<*number*> ...]

mutex [**lock** | **unlock**]

The **mutex** subcommand displays the current status of all known mutual exclusion locks in the process. Mutexes to be listed can be specified through the <*number*> parameter, or all mutexes will be listed. Users can also choose to display only locked or unlocked mutexes by using the **lock** or **unlock** options.

The information listed for each mutex is as follows:

mutex Indicates the symbolic name of the mutex, in the form $m*mutex_number*.

type Indicates the type of the mutex: non-rec (non recursive), recursi (recursive) or fast.

obj_addr Indicates the memory address of the mutex.

lock Indicates the lock state of the mutex: yes if the mutex is locked, no if not.

owner If the mutex is locked, indicates the symbolic name of the user thread which holds the mutex.

Related to this subcommand are the **attribute condition** and **thread** subcommands.

pdbx next Subcommand

next [*number*]

The **next** subcommand runs the application program up to the next source line. The *number* parameter specifies the number of times the subcommand runs. If the *number* parameter is not specified, **next** runs once only.

The difference between this and the **step** subcommand is that if the line contains a call to a procedure or function, **step** will stop at the beginning of that block, while **next** will not.

If you use the **next** subcommand in a multi-threaded application program, all the user threads run during the operation, but the program continues execution until the running thread reaches the specified source line. By default, breakpoints for all threads are ignored during the **next** command. This behavior can be changed using the **$catchbp** set variable. If you wish to step the running thread only, use the **set** command to set the variable *$hold_next*. Setting this variable may result in deadlock, since the running thread may wait for a lock held by one of the blocked threads.

Related to this subcommmand are the **nexti**, **step**, **stepi**, **return**, **cont**, and **set** subcommands.

pdbx nexti Subcommand

nexti [*number*]

The **nexti** subcommand runs the application program up to the next instruction. The *number* parameter specifies the number of times the subcommand will run. If the *number* parameter is not specified, **nexti** runs once only.

The difference between this and the **stepi** subcommand is that if the line contains a call to a procedure or function, **stepi** will stop at the beginning of that block, while **nexti** will not.

If you use the **nexti** subcommand in a multi-threaded application program, all the user threads run during the operation, but the program continues execution until the running thread reaches the specified machine instruction. If you wish to step the running thread only, use the **set** command to set the variable *$hold_next*. Setting this variable may result in deadlock since the running thread may wait for a lock held by one of the blocked threads.

Related to this subcommand are the **next**, **step**, **stepi**, **return**, **cont**, and **set** subcommands.

pdbx on Subcommand

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

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.

Note: The **pdbx** prompt will be presented after all of the tasks in the temporary context have completed the specified command. It is possible using <**Ctrl-c**> followed by the **back** or the **on** command to issue further **pdbx** commands in the original 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.

When you switch context using **on** *context_name*, and the new context has at least one task in the *running* state, a message is displayed stating that at least one task is in the *running* state. Thus, no **pdbx** prompt is displayed until all tasks in this context are in the *debug ready* state.

When you switch to a context where all states are in the *debug ready* state, the **pdbx** prompt is displayed immediately.

At the **pdbx** subset prompt, **on** *context_name* causes one of the following to happen: either a **pdbx** prompt is displayed; or a message is displayed indicating the reason why the **pdbx** prompt will be displayed at a later time. This is generally because one of the tasks is in running state. See “Context Switch when Blocked” on page 17 for more information on the **pdbx** subset prompt.

At a **pdbx** prompt, you cannot use **on** *context_name pdbx_command* if any of the tasks in the specified context are running.

Assume you have an application running as 15 tasks, and the output of the **group list** subcommand lists the existing task groups as:

```
all         0:D    1:U    2:D    3:D    4:D    5:D    6:U    7:D

            8:D    9:D   10:R   11:R   12:R   13:U   14:U

johnny      0:D

jessica     2:D    3:D    8:D

un          1:U    6:U   13:U   14:U

run        10:R   11:R   12:R

deb         2:D    3:D    4:D    5:D    8:D    9:D
```

1. To add a breakpoint for task 0, enter:

   ```
   on johnny stop at 31
   ```

   The **pdbx** debugger responds with:

   ```
   johnny:[0] 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:

   ```
   jessica:[0] 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)
   ```

4. To add a conditional breakpoint for all tasks in the current context, enter:

   ```
   stop at 48 if len < 1
   ```

   The **pdbx** debugger responds with:

   ```
   johnny:[1] stop at "ring.f":48 if len < 1
   ```

5. To view the events that have been set on the task group “jessica”, enter:

   ```
   on jessica status
   ```

   The **pdbx** debugger responds with:

   ```
   jessica:[0] stop in ring
   ```

6. To add a tracepoint for task 2, enter:

   ```
   on 2
   ```

The **pdbx** debugger responds with the prompt:

```
pdbx(2)
```

Then, enter:

```
trace 57
```

The **pdbx** debugger responds with:

```
2:[0] trace "ring.f":57
```

7. To view all of the events that have been set, enter:

   ```
   status all
   ```

   The **pdbx** debugger responds with:

   ```
   2:[0] trace "ring.f":57

   johnny:[0] stop at "ring.f":48

   johnny:[1] stop at "ring.f":56 if len < 1

   jessica:[0] stop in ring
   ```

Related to this subcommand is the **pdbx group** subcommand.

pdbx print Subcommand

print *expression* ...

print *procedure* ([*parameters*])

The **print** subcommand does either of the following:

- Prints the value of a list of expressions, specified by the *expression* parameters.
- Executes a procedure, specified by the *procedure* parameter, and prints the return value of that procedure. Parameters that are included are passed to the procedure.

To display the value of x and the value of y shifted left two bits, enter:

```
print x, y << 2
```

To display the value returned by calling the **sbrk** routine with an argument of 0, enter:

```
print sbrk(0)
```

To display the sixth through the eighth elements of the Fortran character string *a_string*, enter:

```
print &a_string + 5, &a_string + 7/c
```

Related to this subcommand are the **dbx assign** and **call** subcommands, and the **pdbx set** subcommand.

pdbx quit Subcommand

quit

The **quit** subcommand terminates all program tasks, and ends the **pdbx** debugging session. The **quit** subcommand is context insensitive and has no parameters.

Quitting a debug session in attach mode causes the debugger and all the members of the original **poe** application partition to exit.

To exit the **pdbx** debug program, enter:

```
quit
```

pdbx registers Subcommand

registers

The **registers** subcommand displays the values of general purpose registers, system control registers, floating-point registers, and the current instruction register.

Registers can be displayed or assigned to individually by using the following predefined register names:

$r0 through $r31 for the general purpose registers.

$fr0 through $fr31 for the floating point registers.

$sp, $iar, $cr, $link for, respectively, the stack pointer, program counter, condition register, and link register.

By default, the floating-point registers are not displayed. To display the floating-point registers, use the **unset** *$noflregs* command.

Notes:

1. The register value may be set to the 0xdeadbeef hexadecimal value. The 0xdeadbeef hexadecimal value is an initialization value assigned to general purpose registers at process initialization.
2. The **registers** command cannot display registers if the current thread is in kernel mode.

pdbx return Subcommand

return [*procedure*]

The **return** subcommand causes the program to execute until a return to the procedure, specified by the *procedure* parameter, is reached. If the *procedure* parameter is not specified, execution ceases when the current procedure returns.

pdbx search Subcommand

/<*regular_expression*>[***/***]

?<*regular_expression*>[**?**]

The search forward (***/***) or search backward (**?**) subcommands allow you to search in the current source file for the given <*regular_expression*>. Both forms of search wrap around. The previous regular expression is used if no regular expression is given to the current command.

Related to this subcommand is the **regcmp** subroutine.

pdbx set Subcommand

set [*variable*]

set [*variable=expression*]

The **set** subcommand defines a value for the set variable. The value is specified by the *expression* parameter. The set variable is specified by the *variable* parameter. The name of the variable should not conflict with names in the program being debugged. A variable is expanded to the corresponding expression within other commands. If the **set** subcommand is used without arguments, the currently set variables are displayed.

Related to this subcommand is the **unset** subcommand.

pdbx sh Subcommand

sh <*command*>

The **sh** subcommand passes the command specified by the *command* parameter to the shell on the remote task(s) for execution. The **SHELL** environment variable determines which shell is used. The default is the Bourne shell (sh).

Note: The **sh** subcommand with no arguments is not supported.

To run the **ls** command on all tasks in the current context, enter:

```
sh ls
```

To display contents of the *foo.dat* data file on task 1, enter:

```
on 1 cat foo.dat
```

pdbx skip Subcommand

skip [*number*]

The **skip** subcommand continues execution of the program from the current stopping point, ignoring the next breakpoint. If a *number* variable is supplied, **skip** ignores that next amount of breakpoints.

Related to this subcommand is the **cont** subcommand.

pdbx source Subcommand

source *commands_file*

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.

To read **pdbx** subcommands from a file named “jessica”, enter:

```
source jessica
```

Related to this subcommand is the **dbx source** subcommand.

pdbx status Subcommand

status

status all

A list of **pdbx** events (breakpoints and tracepoints) can be displayed by using the **status** subcommand. You can specify “all” after this command to list all events (breakpoints and tracepoints) that have been set in all groups and tasks. This is valid at the **pdbx** prompt and the **pdbx** subset prompt.

Because the **status** command without “all” specified is context sensitive, it will not display status for events outside the context.

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.

To see an example of **status all**, enter:

```
status all
```

The **pdbx** debugger responds with:

```
0:[0] trace foo at "foo.c":21

1:[0] stop in func

all:[0] stop at "foo.c":19

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

Related to this subcommand are the **pdbx stop**, **trace**, and **delete** subcommands.

pdbx step Subcommand

step [*number*]

The **step** subcommand runs source lines of the program. You specify the number of lines to be executed with the *number* parameter. If this parameter is omitted, the default is a value of 1.

The difference between this and the **next** subcommand is that if the line contains a call to a procedure or function, **step** will enter that procedure or function, while **next** will not.

If you use the **step** subcommand on a multi-threaded program, all the user threads run during the operation, but the program continues execution until the interrupted thread reaches the specified source line. By default, breakpoints for all threads are ignored during the **step** command. This behavior can be changed using the **$catchbp** set variable.

If you wish to step the interrupted thread only, use the **set** subcommand to set the variable *$hold_next*. Setting this variable may result in debugger induced deadlock, since the interrupted thread may wait for a lock held by one of the threads blocked by *$hold_next*.

Note: Use the *$stepignore* variable of the **set** subcommand to control the behavior of the **step** subcommand. The *$stepignore* variable enables **step** to step over large routines for which no debugging information is available.

Related to this subcommand are the **stepi**, **next**, **nexti**, **return**, **cont**, and **set** commands.

pdbx stepi Subcommand

stepi [*Number*]

The **stepi** subcommand runs instructions of the program. You specify the number of instructions to be executed with the *number* parameter. If the parameter is omitted, the default is 1.

If used on a multi-threaded program, the **stepi** subcommand steps the interrupted thread only. All other user threads remain stopped.

Related to this subcommand are the **step**, **next**, **nexti**, **return**, **cont**, and **set** subcommands.

pdbx stop Subcommand

stop if <*condition*>
stop at <*source_line_number*> [**if** <condition>]
stop in <*procedure*> [**if** <condition>]
stop <*variable*> [**if** <condition>]
stop <*variable*> **at** <*source_line_number*>
[**if** <condition>]
stop <*variable*> **in** <*procedure*> [**if** <condition>]

Specifying **stop at** <*source_line_number*> causes the breakpoint to be triggered each time that source line is reached.

Specifying **stop in** <*procedure*> causes the breakpoint to be triggered each time the program counter reaches the first executable source line in the procedure (function, subroutine).

Using the <*variable*> argument to stop causes the breakpoint to be triggered when the contents of the variable changes. This form of breakpoint can be very time consuming. For better results, when possible, further qualify these breakpoints with a *source_line* or *procedure* argument.

Specify the <*condition*> argument using the syntax described by "Specifying Expressions" on page 36.

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.

Specifying **stop at** <*source_line_number*> causes the breakpoint to be triggered each time that source line is reached.

Specifying **stop in** <*procedure*> causes the breakpoint to be triggered each time the program counter reaches the first executable source line in the procedure (function, subroutine).

Using the <*variable*> argument to stop causes the breakpoint to be triggered when the contents of the variable changes. This form of breakpoint can be very time consuming. For better results, when possible, further qualify these breakpoints with a *source_line* or *procedure* argument.

Specify the <*condition*> argument using the syntax described by "Specifying Expressions" on page 36.

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.

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 to this subcommand are the **dbx stop** and **which** subcommands, and the **pdbx trace**, **status**, and **delete** subcommands.

pdbx tasks Subcommand

tasks [long]

With the **tasks** subcommand, you display information about all the tasks in the partition. Task state information is always displayed. If you specify “long” after the command, it also displays the name, ip address, and job manager number associated with the task.

Following is an example of output produced by the **tasks** and **tasks long** command.

```
pdbx(others) tasks

  0:D     1:D     2:U     3:U     4:R     5:D     6:D     7:R



pdbx(others) tasks long

  0:Debug ready   pe04.kgn.ibm.com                 9.117.8.68       -1

  1:Debug ready   pe03.kgn.ibm.com                 9.117.8.39       -1

  2:Unhooked      pe02.kgn.ibm.com                 9.117.11.56      -1

  3:Unhooked      augustus.kgn.ibm.com             9.117.7.77       -1

  4:Running       pe04.kgn.ibm.com                 9.117.8.68       -1

  5:Debug ready   pe03.kgn.ibm.com                 9.117.8.39       -1

  6:Debug ready   pe02.kgn.ibm.com                 9.117.11.56      -1

  7:Running       augustus.kgn.ibm.com             9.117.7.77       -1
```

Related to this subcommand is the **pdbx group** subcommand.

pdbx thread Subcommand

thread
thread [<*number*>...]
thread [**info**] [<*number*> ...]
thread [**run | wait | susp | term**]
thread [**hold | unhold**] [<*number*> ...]
thread [**current**] [<*number*>]

The **thread** subcommand displays the current status of all known threads in the process. Threads to be displayed can be specified through the <*number*> parameters, or all threads will be listed. Threads can also be selected by states using the **run**, **wait**, **susp**, **term, or current** options. The **info** option can be used to display full information about a thread. The **hold** and **unhold** options affect whether the thread is dispatchable when further execution control commands are issued. A thread that has been held will not be given any execution time until the unhold option is issued. The **thread** subcommand displays a column indicating whether a thread is held or not. No further execution will occur if the interrupted thread is held.

The information displayed by the **thread** subcommand is as follows:

thread Indicates the symbolic name of the user thread, in the form $t*thread_number*.

state-k Indicates the state of the kernel thread (if the user thread is attached to a kernel thread). This can be run, wait, susp, or term, for running, waiting, suspended, or terminated.

wchan Indicates the event on which the kernel thread is waiting or sleeping (if the user thread is attached to a kernel thread).

state-u Indicates the state of the user thread. Possible states are running, blocked, or terminated.

k-tid Indicates the kernel thread identifier (if the user thread is attached to a kernel thread).

mode Indicates the mode (kernel or user) in which the user thread is stopped (if the user thread is attached to a kernel thread).

held Indicates whether the user thread has been held.

scope Indicates the contention scope of the user thread; this can be sys or pro for system or process contention scope.

function Indicates the name of the user thread function.

The displayed thread (“>”) is the thread that is used by other **pdbx** commands that are thread specific such as:

down

dump

file

func

list

listi

print

registers

up

where

The displayed thread defaults to be the interrupted thread after each execution control command. The displayed thread can be changed using the current option.

The interrupted thread ("*") is the thread that stopped first and because it stopped, in turn caused all of the other threads to stop. The interrupted thread is treated specially by subsequent **step**, **next**, and **nexti** commands. For these stepping commands, the interrupted thread is stepped, while all other (unheld) threads are allowed to continue.

To force only the interrupted thread to execute during execution control commands, set the *$hold_next* set variable. Note that this can create a debugger induced deadlock if the interrupted thread blocks on one of the other threads.

Note that the **pdbx** documentation uses "interrupted thread" in the same way the **dbx** documentation uses "running thread." Also, the **pdbx** documentation uses "displayed thread" in the same way the **dbx** documentation uses "current thread."

Related to this subcommand are the **attribute condition** and **mutex** subcommands.

pdbx trace Subcommand

trace [**in** <*procedure*>] [**if** <condition>]
trace <*source_line_number*> [**if** <condition>]
trace <*procedure*>
[**in** <*procedure*>]
[**if** <condition>]
trace <*variable*> [**in** <*procedure*>]
[**if** <condition>]
trace <*expression*> **at** <*source_line_number*>
[**if** <condition>]

Specifying **trace** with no arguments causes trace information to be displayed for every source line in your program.

Specifying **trace** <*source_line_number*> causes the tracepoint to be triggered each time that source line is reached.

Specifying **trace** [**in** <*procedure*>] causes the tracepoint to be triggered each time your program executes a source line within the procedure (function, subroutine).

Using the <*variable*> argument to trace causes the tracepoint to be triggered when the contents of the variable changes. This form of tracepoint can be very time consuming. For better results, when possible, further qualify these tracepoints with a *source_line* or *procedure* argument.

Specify the <*condition*> argument using the syntax described by "Specifying Expressions" on page 36.

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.

Specifying **trace** with no arguments causes trace information to be displayed for every source line in your program.

Specifying **trace** <*source_line_number*> causes the tracepoint to be triggered each time that source line is reached.

Specifying **trace** [**in** <*procedure*>] causes the tracepoint to be triggered each time your program executes a source line within the procedure (function, subroutine).

Using the <*variable*> argument to trace causes the tracepoint to be triggered when the contents of the variable changes. This form of tracepoint can be very time consuming. For better results, when possible, further qualify these tracepoints with a *source_line* or *procedure* argument.

Specify the <*condition*> argument using the syntax described by “Specifying Expressions” on page 36.

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.

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 to this subcommand are the **dbx trace** and **which** subcommands, and the **pdbx stop**, **status**, and **delete** subcommands.

pdbx unalias Subcommand

unalias *alias_name*

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.

To remove the alias “p”, enter:

```
unalias p
```

Related to this subcommand is the **pdbx alias** subcommand.

pdbx unhook Subcommand

unhook

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.

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:D    1:U    2:D    3:D
```

Tasks marked with the letter U next to them are unhooked tasks. In this case, task 1 is unhooked. Tasks marked with the letter D are debug ready, hooked tasks. In this case, tasks 0, 2, and 3 are hooked.

Related to this subcommand is the **dbx detach** subcommand and the **pdbx hook** subcommand.

pdbx unset Subcommand

unset *name*

The **unset** subcommand removes the set variable associated with the specified *name*.

Related to this subcommand is the **set** subcommand.

pdbx up Subcommand

up [*count*]

The **up** subcommand moves the current function up the stack the number of levels you specify with the *count* parameter. The current function is used for resolving names. The default for the *count* parameter is 1.

The **up** and **down** subcommands can be used to navigate through the call stack. Using these subcommands to change the current function also causes the current file and local variables to be updated to the chosen stack level.

Related to this subcommand are the **down**, **print**, **dump**, **func**, **file**, and **where** subcommands.

pdbx use Subcommand

use [*directory ...*]

The **use** subcommand sets the list of directories to be searched when the **pdbx** debugger looks for source files. If the subcommand is specified without arguments, the current list of directories to be searched is displayed.

The @ (at sign) is a special symbol that directs **pdbx** to look at the full path name information in the object file, if it exists. If you have a relative directory called @ to search, you should use ./@ in the search path.

The **use** subcommand uses the + (plus sign) to add more directories to the list of directories to be searched. If you have a directory named +, specify the full path name for the directory (for example, ./+ or /tmp/+).

Related to this subcommand are the **file** and **list** subcommands.

pdbx whatis Subcommand

whatis <*name*>

The **whatis** subcommand displays the declaration of what you specify as the *name* parameter. The *name* parameter can designate a variable, procedure, or function name, optionally qualified with a block name.

Related to this subcommand are the **whereis** and **which** subcommands.

pdbx where Subcommand

where

The **where** subcommand displays a list of active procedures and functions. For example:

```
pdbx(all) where

init_trees(), line 23 in "funcs5.c"

colors(depth = 30, str = "This is it"), line 61 in "funcs5.c"

newmain(), line 59 in "funcs2.c"

f6(), line 25 in "funcs2.c"

main(argc = 1, argv = 0x2ff21c58), line 125 in "funcs.c"
```

Related to this subcommand are the **dbx up** and **down** subcommands.

pdbx whereis Subcommand

whereis *identifier*

The **whereis** subcommand displays the full qualifications of all the symbols whose names match the specified *identifier*. The order in which the symbols print is not significant.

Related to this subcommand are the **whatis** and **which** commands.

pdbx which Subcommand

which *identifier*

The **which** subcommand displays the full qualification of the given *identifier*. The full qualification consists of a list of the outer blocks with which the *identifier* is associated.

Related to this subcommand are the **whatis** and **whereis** subcommands.

pedb

NAME

pedb – Invokes the **pedb** debugger, which is the X-Windows interface of the PE debugging facility.

SYNOPSIS

pedb [[*program*] *program options*] [*poe options*] [*X options*]
[[**-I** *source directory*]...]
[**-d** *nesting depth*]
[**-x**]

pedb -a *poe process id* [*limited poe options*] [*X options*]
[[**-I** *source directory*]...]
[**-d** *nesting depth*]
[**-x**]

pedb -h

The **pedb** command invokes the **pedb** debugger, which is the X-Windows interface of the PE debugging facility.

FLAGS

The **pedb** command accepts standard X-Windows flags. Because the **pedb** 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:

-a Attaches to a running **poe** job by specifying its process id. This must be executed from the node where the **poe** job was initiated. When using the debugger in attach mode there are some debugger command line arguments that should not be used. In general, any arguments that control how the partition is set up or specify application names and arguments should not be used.

-d Sets the limit for the nesting of program blocks. The default nesting depth limit is 25.

-h Writes the **pedb** usage to STDERR. This includes **pedb** command line syntax and a description of **pedb** flags.

-I (upper-case i)
Specifies a directory to be searched for an executable's source files. This flag must be specified multiple times to set multiple paths. (Once **pedb** is running, this list can also be updated using the Update Source Path window.)

-x Prevents stripping _ (trailing underscore) characters from symbols originating in Fortran source code. This flag enables distinguishing between symbols which are identical except for an underscore character, such as xxx and xxx_.

DESCRIPTION

The **pedb** command invokes the X-Windows interface of the PE debugging facility. It runs in the Parallel Operating Environment.

To use **pedb** 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. It is also advisable to not use the optimization option, **-O**. Using the debugger on optimized code may produce inconsistent and erroneous results. For more information on the **-g** and **-O** compiler options, refer to their use on other compiler commands such as **cc** and **xlf**. These compiler commands are described in *IBM AIX Version 4 Commands Reference*

ENVIRONMENT VARIABLES

Because the **pedb** command runs in the Parallel Operating Environment, it interacts with most of the same environment variables associated with the **poe** command. See the **poe** manual page in *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for a description of these environment variables. As indicated by the syntax statements, you are also able to specify **poe** command line options when invoking **pedb**. Using these options will override the setting of the corresponding environment variable, as is the case when invoking a parallel program with the **poe** command.

In conjunction with **pedb** array visualization, you can set the **MP_DEBUG_BIN_DIR** evironment variable to customize this feature. See Appendix D, “Visualization Customization and Data Explorer Samples” on page 243 for more information.

EXAMPLES

To start the **pedb** debugger, enter:

```
pedb weather temperate asia -procs 16 -labelio yes
```

This will invoke **pedb** running the weather application on a partition containing 16 nodes with all program output labeled by task id.

The **pedb** window automatically opens to mark the start of the debug session.

FILES

host.list (Default host list file)

/usr/lib/X11/app-defaults/Pedb (Xdefaults file)

RELATED INFORMATION

Commands: **pdbx**(1), **poe**(1), **mpxlf**(1), **mpcc**(1), **cc**(1), **xlf**(1) **mpxlf_r**(1) **mpcc_r**(1) **mpCC_r**(1)

vt

NAME

vt – Starts the Visualization Tool, which is an X-Windows tool that enables you to visualize performance characteristics of your partition, or playback traces generalized from a POE program.

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 online 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 SP 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 online performance monitoring of a cluster or mixed environment, 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.

-go Starts playing back the trace file immediately upon starting VT. When you use this flag, you must also specify a trace file and a configuration file using the **-tracefile** (or **-tfile**) flag and the **-configfile** (or **-cfile**) flags.

-log_file Specifies the file name where the results of the trace file post-processing will be written. The default name is **$HOME/tracefilename.pplog**.

-h, -?, or -help Gets help information.

-mp_source Specifies which task’s source code is displayed in the Source Code view.

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 online 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 online monitor showing the operational status and activity of each of the processor nodes on an SP system or RS/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 *digd* 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 *digd* 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 SP system Resource Manager is not available:

```
vt -norm
```

FILES

host.list (Default host list file)

/usr/lib/X11/app-defaults/Vt (Xdefaults file)

$HOME/tracefilename.pplog (Default file name where the results of the trace file post-processing will be written)

RELATED INFORMATION

Commands: **poestat**(1)

Appendix B. Command Line Flags for Normal or Attach Mode

This appendix lists the command line flags that **poe** and **pedb** use, indicating which ones are valid in normal and in attach debugging mode. When starting in attach mode, the debugger gives a message listing the invalid flags used, and then exits.

Table 14 (Page 1 of 2). Command Line Flags for Normal or Attach Mode

Flag	Description	Normal Mode	Attach Mode
-procs	number of processors	yes	no
-tmpdir	temp output directory	no	no
-hostfile	name of host list file	yes	no
-hfile	name of host list file	yes	no
-tracefile	name of trace file	no	no
-tfile	name of trace file	no	no
-tracedir	name of trace directory	no	no
-tdir	name of trace directory	no	no
-infolevel	message reporting level	yes	yes
-ilevel	message reporting level	yes	yes
-tracelevel	trace reporting level	no	no
-tlevel	trace reporting level	no	no
-retry	wait for processors	yes	no
-pmlights	number of LEDs	yes	no
-usrport	port for API-to-user programmable monitor	yes	no
-samplefreq	sampling frequency	no	no
-sfreq	sampling frequency	no	no
-tbuffwrap	wraparound trace buffer	no	no
-tbwrap	wraparound trace buffer	no	no
-tbuffsize	trace buffer size	no	no
-tbsize	trace buffer size	no	no
-ttempsize	trace temp filesize	no	no
-ttsize	trace temp filesize	no	no
-resd	directive to use Resource Manager	yes	no
-euilib	eui library to use	yes	no
-euidevice	adapter set to use for message passing - either Ethernet, FDDI, token ring, or the RS/6000 SP' high-performance communication adapter	yes	no
-euidevelop	EUI develop mode	yes	no
-vtlibpath	VT tracing library	no	no
-newjob	submit new PE jobs without exiting PE	no	no
-pmdlog	use pmd logfile	yes	yes
-savehostfile	list of hosts from resource manager	yes	no
-cmdfile	PE command file	no	no
-stdoutmode	STDOUT mode	yes	no
-stdinmode	STDIN mode	yes	no
-labelio	label output	yes	yes - debugger only
-euilibpath	eui library path	yes	no

Table 14 (Page 2 of 2). Command Line Flags for Normal or Attach Mode

Flag	**Description**	**Normal Mode**	**Attach Mode**
-pgmmodel	programming model	no	no
-retrycount	retry count for node allocation	yes	no
-rmpool	default pool for job manager	yes	no
-spname	hostname of SP for jm_connect	yes	no
-cpu_use	cpu usage	yes	no
-adapter_use	adapter usage	yes	no
-pulse	poe pulse	no	no
-d	nesting depth of program blocks	yes	yes
-I (upper case i)	path to search for source files	yes	yes
-x	prevents the dbx command from stripping trailing underscore in Fortran	yes	yes
-a	start in attach mode	N/A	yes

Appendix C. Exporting Arrays to Hierarchical Data Format (HDF)

HDF is a multi-object data format from the National Center of Excellence in Supercomputing Applications (NCSA), which is used for scientific and visualization data. **Pedb** uses HDF (version 3.3) as the data format for exporting arrays in both the array export and visualization features. See "Exporting Array Information to File" on page 81 and "Visualizing Program Arrays" on page 97 for more information.

HDF supports a variety of object types within a file. **Pedb** exports array information using the Scientific Data Set (SDS) type plus additional array attributes. If you are exporting more than one array to the export file concurrently, **pedb** will create one SDS for each array.

The following tables list the attributes written to the HDF file by **pedb**.

The first table gives dimension attributes; one for each dimension of each SDS.

Table 15. Array attributes defined by pedb

Name	Type	Values	Description
Label	Dimension	Any string (no default)	Label that describes this dimension
Units	Dimension	Any string (no default)	Units to be used with this dimension
Format	Dimension	Any string (no default)	Format to be used to display scale

The next table gives SDS attributes; one for each SDS.

Table 16 (Page 1 of 2). Array attributes defined by pedb

Name	Type	Values	Description
Variable Name	File	Any string (defaults to the array declaration)	This is the string defined by the user on the Export Dialog window
Language_Type	SDS	"C" or "F77"	The source language from which the array was exported
Datetime	SDS	32-bit signed integer	The time the SDS was created
Bele	SDS	Always 1 (Big Endian = 1 , Little Endian = 0)	Big Endian or Little Endian
Attached_ID	SDS	-1 (always)	Reserved - not used
SamplingFormat	SDS	1 (always)	Reserved - not used
Attach_Type	SDS	"RS6000"	String indicating processor type
Task_ID	SDS	32-bit signed integer	Task id of the task where the array was exported
Process_ID	SDS	32-bit signed integer	Process id of the task where the array was exported
Host	SDS	<hostname string>	String name of the host where the array was exported

Table 16 (Page 2 of 2). Array attributes defined by pedb

Name	**Type**	**Values**	**Description**
SamplingStart	SDS	32-bit signed integer(s)	An array of minimum values of the subrange for each dimension
SamplingStride	SDS	32-bit signed integer(s)	An array of strides of the subrange for each dimension
SamplingEnd	SDS	32-bit signed integer(s)	An array of maximum values of the subrange for each dimension
BaseMinimum	SDS	32-bit signed integer(s)	An array of the base minimum of the subrange for each dimension.
BaseMaximum	SDS	32-bit signed integer(s)	An array of the base maximum of the subrange for each dimension
MemAddress	SDS	32-bit signed integer	The base address of the array that was exported

Appendix D. Visualization Customization and Data Explorer Samples

Included as part of the **pedb** debugger array visualization feature is a set of prepackaged sample visualization interfaces to IBM's Visualization Data Explorer (DX).

These interfaces are provided as a set of prepackaged tools that can be:

- used directly by **pedb** without modification
- used along with source code so you can modify them to fit your specific needs.

The Data Explorer samples use the DXLink feature of Version 3.1 of IBM's Visualization Data Explorer. For additional information on IBM's Visualization Data Explorer and DXLink, see the following references:

- *IBM Visualization Data Explorer: Programmer's Reference (Fourth Edition)* SC38-0497-03
- *IBM Visualization Data Explorer: Quick Start Guide (Second Edition)* SC34-3262-01
- *IBM Visualization Data Explorer: User's Reference (Second Edition)* SC38-0486-01
- *IBM Visualization Data Explorer: User's Guide (Fifth Edition)* SC38-0496-04

The sample interfaces can be found in the **pedb** samples directory */usr/lpp/ppe.pedb/samples*. The following table describes the files included in the directory.

Name	Type	Description
DX Files	v*.net v*.cfg	Data Explorer visual nets Data Explorer configuration files for each of the visual nets
DXLink files	DXLvisual.c DXLvisual_Motif.c	Command line version Motif version
Makefile	makefile.DXL	Makefile to build the DXLink files
Executables	DXLvisual DXLvisual_Motif	The executables are installed and used as part of the integrated prepackaged visualizations in **pebd**.

You may wish to modify the samples in a variety of ways. Some examples include:

- Modifying the DX nets
- Enhancing the DXLink programs to use more advanced features
- Replacing the the DX samples with your own visualization tools.

Here is some information you may find useful when making these modifications.

- The visualization feature of **pedb** is designed to export the selected array to a temporary file first, and then pass the name of the temporary file to a Korn shell script for execution.
- Each Visualization type on the **pedb** Visualization Dialog Window has a corresponding Korn shell script associated with it. These are located in */usr/lpp/ppe.pedb/bin*.
- Each Visualization type on the **pedb** Visualization Dialog Window has a corresponding entry in the X defaults file that defines its label.
- The default location of the shell scripts can be overridden using the **MP_DEBUG_BIN_DIR** environment variable. This allows you to override individual scripts without involvement of the system administrator.
- Each of the prepackaged scripts call **DXLvisual_Motif**, passing it the temporary file name and the DX visual program (*.net*) to execute.

There are three major points of customization that are available:

1. Modify the Korn shell scripts in */usr/lpp/ppe.pedb/bin* to call an entirely different visualization program or a different DX visual net.
2. Enhance the **DXLvisual_Motif** or **DXLvisual** programs to take advantage of more advanced features of DX. A makefile (*makefile.DXL*) is included to rebuild these programs.
3. Modify the DX visual nets to perform a custom visualization.

Note: The *Pedb.ad* for the Visualization Dialog menu label entry should be updated to be consistent with these changes.

The following steps give an example of a visualization customization.

1. Copy shell script *VisualTypeOption_1.ksh* from */usr/lpp/ppe.pebd/bin* into your current working directory.

   ```
   $ cp /usr/lpp/ppe.pedb/bin/VisualTypeOption_1.ksh
   ```

2. Edit the script to pass a different DX visualization, *v7.net*.

   ```
   <old>


   $MP_DEBUG_SAMPLE_DIR/DXLvisual_Motif $MP_DEBUG_SAMPLE_DIR/v1.net $1 -geometry +0+0


   <new>


   $MP_DEBUG_SAMPLE_DIR/DXLvisual_Motif $MP_DEBUG_SAMPLE_DIR/v7.net $1 -geometry +0+0
   ```

3. Set the **MP_DEBUG_BIN_DIR** environment to allow **pebd** to find your custom version of the script.

   ```
   $ export MP_DEBUG_BIN_DIR=.
   ```

   Note: This will only override the script you have copied to your local directory.

4. Find the label entry in */usr/lpp/ppe.pedb/defaults/Pedb.ad* to copy to your local *.Xdefaults* file.

```
Pedb*VisualTypeOption_1.labelString:           DX 2D Colormap

Pedb*VisualTypeOption_1.mnemonic:              C
```

5. Edit the label entry to display you own custom label.

```
Pedb*VisualTypeOption_1.labelString:           My Custom Visual

Pedb*VisualTypeOption_1.mnemonic:              M
```

6. Try out your new visualization within **pedb**.

Note: Make sure that the Korn shell script is set to be executable.

Appendix E. Customizing Tool 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 **pedb** 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 *Pedb* for the Parallel Environment debugger and *Vt* for the Visualization Tool. 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
```

Customizable resources and instructions for their use for **pedb** are defined in **/usr/lpp/ppe.pedb/defaults/Pedb.ad**. In this file is a set of X resources for defining graphical user interfaces based on the following criteria:

- Window geometry
- Push button and label text

- Pixmaps.

Customizable resources and instructions for their use for VT are defined in **/usr/lpp/ppe.vt/defaults/Vt**. In this file is a set of X resources for defining graphical user interfaces based on the following criteria:

- Window geometry
- Push button and label text
- Pixmaps.

Glossary of Terms and Abbreviations

This glossary includes terms and definitions from:

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

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

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

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

A

address. A value, possibly a character or group of characters that identifies a register, a device, a particular part of storage, or some other data source or destination.

AIX. Abbreviation for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system. AIX is particularly suited to support technical computing applications, including high function graphics and floating point computations.

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

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

API. Application Programming Interface.

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

argument. A parameter passed between a calling program and a called program or subprogram.

attribute. A named property of an entity.

B

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

blocking operation. An operation which does not complete until the operation either succeeds or fails. For example, a blocking receive will not return until a message is received or until the channel is closed and no further messages can be received.

breakpoint. A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

broadcast operation. A communication operation in which one processor sends (or broadcasts) a message to all other processors.

buffer. A portion of storage used to hold input or output data temporarily.

C

C. A general purpose programming language. It was formalized by ANSI standards committee for the C language in 1984 and by Uniforum in 1983.

C++. A general purpose programming language, based on C, which includes extensions that support an object-oriented programming paradigm. Extensions include:

- strong typing
- data abstraction and encapsulation
- polymorphism through function overloading and templates
- class inheritance.

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

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

client. A function that requests services from a server, and makes them available to the user.

cluster. A group of processors interconnected through a high speed network that can be used for high performance computing. It typically provides excellent price/performance.

collective communication. A communication operation which involves more than two processes or tasks. Broadcasts, reductions, and the MPI_Allreduce subroutine are all examples of collective communication operations. All tasks in a communicator must participate.

command alias. When using the PE command line debugger, pdbx, you can create abbreviations for existing commands using the **pdbx alias** command. These abbreviations are know as *command aliases*.

Communication Subsystem (CSS). A component of the IBM Parallel System Support Programs for AIX that provides software support for the High Performance Switch. It provides two protocols; IP (Internet Protocol) for LAN based communication and US (user space) as a message passing interface that is optimized for performance over the switch. See also *Internet Protocol* and *User Space*.

communicator. An MPI object that describes the communication context and an associated group of processes.

compile. To translate a source program into an executable program.

condition. One of a set of specified values that a data item can assume.

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

core dump. A process by which the current state of a program is preserved in a file. Core dumps are usually associated with programs that have encountered an unexpected, system-detected fault, such as a Segmentation Fault, or severe user error. The current program state is needed for the programmer to diagnose and correct the problem.

core file. A file which preserves the state of a program, usually just before a program is terminated for an unexpected error. See also *core dump*.

current context. When using either of the PE parallel debuggers, control of the parallel program and the display of its data can be limited to a subset of the tasks that belong to that program. This subset of tasks is called the *current context*. You can set the current context to be a single task, multiple tasks, or all the tasks in the program.

D

data decomposition. A method of breaking up (or decomposing) a program into smaller parts to exploit parallelism. One divides the program by dividing the data (usually arrays) into smaller parts and operating on each part independently.

data parallelism. Refers to situations where parallel tasks perform the same computation on different sets of data.

dbx. A symbolic command line debugger that is often provided with UNIX systems. The PE command line debugger, **pdbx**, is based on the **dbx** debugger.

debugger. A debugger provides an environment in which you can manually control the execution of a program. It also provides the ability to display the program's data and operation.

distributed shell (dsh). An IBM Parallel System Support Programs for AIX command that lets you issue commands to a group of hosts in parallel. See the *IBM RISC System/6000 Scalable POWERparallel Systems: Command and Technical Reference* (GC23-3900-00) for details.

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

E

environment variable. 1. A variable that describes the operating environment of the process. Common environment variables describe the home directory, command search path, and the current time zone. 2. A variable that is included in the current software environment and is therefore available to any called program that requests it.

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

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

executable. A program that has been link-edited and therefore can be run in a processor.

execution. To perform the actions specified by a program or a portion of a program.

expression. In programming languages, a language construct for computing a value from one or more operands.

F

fairness. A policy in which tasks, threads, or processes must be allowed eventual access to a resource for which they are competing. For example, if multiple threads are simultaneously seeking a lock, then no set of circumstances can cause any thread to wait indefinitely for access to the lock.

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

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

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

foreign host. See *remote host*.

Fortran. One of the oldest of the modern programming languages, and the most popular language for scientific and engineering computations. It's name is a contraction of *FORmula TRANslation*. The two most common Fortran versions are Fortran 77, originally standardized in 1978, and Fortran 90. Fortran 77 is a proper subset of Fortran 90.

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

function cycle. A chain of calls in which the first caller is also the last to be called. A function that calls itself recursively is not considered a function cycle.

functional decomposition. A method of dividing the work in a program to exploit parallelism. One divides the program into independent pieces of functionality which are distributed to independent processors. This is in contrast to data decomposition which distributes the same work over different data to independent processors.

functional parallelism. Refers to situations where parallel tasks specialize in particular work.

G

Gauss-Seidel. An iterative relaxation method for solving Laplace's equation. It calculates the general solution by finding particular solutions to a set of discrete points distributed throughout the area in question. The values of the individual points are obtained by averaging the values of nearby points. Gauss-Seidel differs from Jacobi-Seidel in that for the i+1st iteration Jacobi-Seidel uses only values calculated in the ith iteration. Gauss-Seidel uses a mixture of values calculated in the ith and i+1st iterations.

global max. The maximum value across all processors for a given variable. It is global in the sense that it is global to the available processors.

global variable. A variable defined in one portion of a computer program and used in at least one other portion of the computer program.

gprof. A UNIX command that produces an execution profile of C, Pascal, Fortran, or COBOL programs. The execution profile is in a textual and tabular format. It is useful for identifying which routines use the most CPU time. See the man page on **gprof**.

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

H

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

HIPPI. High performance parallel interface.

hook. **hook** is a **pdbx** command that allows you to re-establish control over all task(s) in the current context that were previously unhooked with this command.

home node. The node from which an application developer compiles and runs his program. The home node can be any workstation on the LAN.

host. A computer connected to a network, and providing an access method to that network. A host provides end-user services.

host list file. A file that contains a list of host names, and possibly other information, that was defined by the application which reads it.

host name. The name used to uniquely identify any computer on a network.

hot spot. A memory location or synchronization resource for which multiple processors compete excessively. This competition can cause a disproportionately large performance degradation when one processor that seeks the resource blocks, preventing many other processors from having it, thereby forcing them to become idle.

I

IBM Parallel Environment for AIX. A program product that provides an execution and development environment for parallel Fortran, C, or C++ programs. It also includes tools for debugging, profiling, and tuning parallel programs.

installation image. A file or collection of files that are required in order to install a software product on a RS/6000 workstation or on SP system nodes. These files are in a form that allows them to be installed or removed with the AIX **installp** command. See also *fileset*, *Licensed Program Product*, and *package*.

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

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

IP. See *Internet Protocol*.

J

Jacobi-Seidel. See *Gauss-Seidel*.

| **job management system**.

| The software you use to manage the jobs across your
| system, based on the availability and state of system
| resources.

K

Kerberos. A publicly available security and authentication product that works with the IBM Parallel System Support Programs for AIX software to authenticate the execution of remote commands.

kernel. The core portion of the UNIX operating system which controls the resources of the CPU and allocates them to the users. The kernel is memory-resident, is said to run in *kernel mode* (in other words, at higher execution priority level than *user mode*) and is protected from user tampering by the hardware.

L

Laplace's equation. A homogeneous partial differential equation used to describe heat transfer, electric fields, and many other applications.

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

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

LoadLeveler. A job management system that works with POE to allow users to run jobs and match processing needs with system resources, in order to better utilize the system.

local variable. A variable that is defined and used only in one specified portion of a computer program.

loop unrolling. A program transformation which makes multiple copies of the body of a loop, placing the copies also within the body of the loop. The loop trip count and index are adjusted appropriately so the new loop computes the same values as the original. This transformation makes it possible for a compiler to take additional advantage of instruction pipelining, data cache effects, and software pipelining.

See also *optimization*.

M

menu. A list of options displayed to the user by a data processing system, from which the user can select an action to be initiated.

message catalog. A file created using the AIX Message Facility from a message source file that contains application error and other messages, which can later be translated into other languages without having to recompile the application source code.

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

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

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

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

N

network. An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

node. (1) In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network. (2) In terms of the RS/6000 SP, a single location or workstation in a network. An SP node is a physical entity (a processor).

node ID. A string of unique characters that identifies the node on a network.

nonblocking operation. An operation, such as sending or receiving a message, which returns immediately whether or not the operation was completed. For example, a nonblocking receive will not wait until a message is sent, but a blocking receive will wait. A nonblocking receive will return a status value that indicates whether or not a message was received.

O

object code. The result of translating a computer program to a relocatable, low-level form. Object code contains machine instructions, but symbol names (such as array, scalar, and procedure names), are not yet given a location in memory.

optimization. A not strictly accurate but widely used term for program performance improvement, especially for performance improvement done by a compiler or other program translation software. An optimizing compiler is one that performs extensive code transformations in order to obtain an executable that runs faster but gives the same answer as the original. Such code transformations, however, can make code debugging and performance analysis very difficult because complex code transformations obscure the correspondence between compiled and original source code.

option flag. Arguments or any other additional information that a user specifies with a program name. Also referred to as *parameters* or *command line options*.

P

package. A number of filesets that have been collected into a single installable image of program products, or LPPs. Multiple filesets can be bundled together for installing groups of software together. See also *fileset* and *Licensed Program Product*.

parallelism. The degree to which parts of a program may be concurrently executed.

parallelize. To convert a serial program for parallel execution.

Parallel Operating Environment (POE). An execution environment that smooths the differences between serial and parallel execution. It lets you submit and manage parallel jobs. It is abbreviated and commonly known as POE.

parameter. * (1) In Fortran, a symbol that is given a constant value for a specified application. (2) An item in a menu for which the operator specifies a value or for which the system provides a value when the menu is interpreted. (3) A name in a procedure that is used to refer to an argument that is passed to the procedure. (4) A particular piece of information that a system or application program needs to process a request.

partition. (1) A fixed-size division of storage. (2) In terms of the RS/6000 SP, a logical definition of nodes to be viewed as one system or domain. System partitioning is a method of organizing the SP into groups of nodes for testing or running different levels of software of product environments.

Partition Manager. The component of the Parallel Operating Environment (POE) that allocates nodes, sets up the execution environment for remote tasks, and manages distribution or collection of standard input (STDIN), standard output (STDOUT), and standard error (STDERR).

pdbx. **pdbx** is the parallel, symbolic command line debugging facility of PE. **pdbx** is based on the **dbx** debugger and has a similar interface.

PE. The IBM Parallel Environment for AIX program product.

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

POE. See Parallel Operating Environment.

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

point-to-point communication. A communication operation which involves exactly two processes or tasks. One process initiates the communication through a *send* operation. The partner process issues a *receive* operation to accept the data being sent.

procedure. (1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a procedure call. (2) A set of related control statements that cause one or more programs to be performed.

process. A program or command that is actually running the computer. It consists of a loaded version of the executable file, its data, its stack, and its kernel data structures that represent the process's state within a multitasking environment. The executable file contains the machine instructions (and any calls to shared objects) that will be executed by the hardware. A process can contain multiple threads of execution.

The process is created via a **fork**() system call and ends using an **exit**() system call. Between **fork** and **exit**, the process is known to the system by a unique process identifier (pid).

Each process has its own virtual memory space and cannot access another process's memory directly. Communication methods across processes include pipes, sockets, shared memory, and message passing.

prof. A utility which produces an execution profile of an application or program. It is useful to identifying which routines use the most CPU time. See the man page for **prof**.

profiling. The act of determining how much CPU time is used by each function or subroutine in a program. The histogram or table produced is called the execution profile.

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

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

R

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

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

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

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

| **Resource Manager**. A server that runs on one of the
| nodes of an RS/6000 SP (SP) machine. It prevents
| parallel jobs from interfering with each other, and
| reports job-related node information.

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

S

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

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

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

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

source line. A line of source code.

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

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

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

standard input (STDIN). In the AIX operating system, the primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

standard output (STDOUT). In the AIX operating system, the primary destination of data produced by a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command.

stencil. A pattern of memory references used for averaging. A 4-point stencil in two dimensions for a given array cell, x(i,j), uses the four adjacent cells, x(i-1,j), x(i+1,j), x(i,j-1), and x(i,j+1).

subroutine. (1) A sequence of instructions whose execution is invoked by a call. (2) A sequenced set of instructions or statements that may be used in one or more computer programs and at one or more points in a computer program. (3) A group of instructions that can be part of another routine or can be called by another program or routine.

synchronization. The action of forcing certain points in the execution sequences of two or more asynchronous procedures to coincide in time.

system administrator. (1) The person at a computer installation who designs, controls, and manages the use of the computer system. (2) The person who is responsible for setting up, modifying, and maintaining the Parallel Environment.

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

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

T

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

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

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

tracepoint. Tracepoints are places in the program that, when reached during execution, cause the debugger to print information about the state of the program.

trace record. In PE, a collection of information about a specific event that occurred during the execution of your program. For example, a trace record is created for each send and receive operation that occurs in your program (this is optional and may not be appropriate). These records are then accumulated into a trace file which allows the Visualization Tool to visually display the communications patterns from the program.

U

unrolling loops. See *loop unrolling*.

US. See *user space*.

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

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

utility program. A computer program in general support of computer processes; for example, a diagnostic program, a trace program, a sort program.

utility routine. A routine in general support of the processes of a computer; for example, an input routine.

V

variable. (1) In programming languages, a named object that may take different values, one at a time. The values of a variable are usually restricted to one data type. (2) A quantity that can assume any of a given set of values. (3) A name used to represent a data item whose value can be changed while the program is running. (4) A name used to represent data whose value can be changed, while the program is running, by referring to the name of the variable.

view. (1) In an information resource directory, the combination of a variation name and revision number that is used as a component of an access name or of a descriptive name.

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

VT. See *Visualization Tool*.

X

X Window System. The UNIX industry's graphics windowing standard that provides simultaneous views of several executing programs or processes on high resolution graphics displays.

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

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

Index

A

adapter 239
address 8, 44
AIX Kernel Statistics trace records 117
allocating nodes 139
application 1
Application Markers trace records 117
Application Message Queues window) 87
argument 21
attribute 82

B

blocking read 18
blocking receive 27, 64
blocking send 26, 53
breakpoint 46
buffer 52

C

C 42
C++ 112
clock synchronization 122
cluster 109
collective communication 117
Collective Communications Details window 94
command alias 2
commands, PE 193
Communication/Program views 149
communicator 153
Computation views 162
condition 60
Connectivity Graph 150
context switches 184
conventions xii
current context 2
customizing resources 247
customizing tool resources 247

D

daemon 117, 236
Data Explorer 98, 243
dbx subcommands 33, 205
debugging parallel programs 1, 41
  with pdbx 1
  with pedb 41
disk reads 174
disk transfers 176
disk views 174
disk writes 178

E

Early Arrival Message Details window 94
Ethernet 239
event 2
executable 5
execution 1
Export File Selection window 85
export options 83
Export window 82
expression 2

F

FDDI 239
file system 123
fileset 98
Find window 103
flag 1
Fortran 5
function 41
  context sensitive subcommands 2

G

global variable 25

H

home node 2
host 241
host list file 5

I

IBM Parallel Environment for AIX xi
instantaneous views 115
Interprocessor Communication 151

K

kernel 111
kernel utilization 162

L

Load Executables window 49
local variable 24

M

Main window 46
menu 58
Message Group Information window) 96
message passing 106
message passing routine 104
message queue debugger 86
  starting the message queue debugger 87
  using the message queue debugger 86
Message Status Matrix 155
mixed system xi
monitoring processor nodes using VT 139, 144
  adjusting sampling interval when 144
  selecting nodes for monitoring 143
  starting monitoring 143
MPI Collective Communication trace records 117
MPI Message Passing trace records 117
MPMD (Multiple Program Multiple Data) 4

N

network 109
Network Time Protocol (NTP) 118
network views 180
node 1
node ID 165
nonblocking calls 151
nonblocking sends 151

O

optimization 1
option 1
Optional Notes window 83

P

packets received 180
packets sent 182
page faults 186
Parallel Operating Environment (POE) xi
parallel programs 1
  compiling for VT 120
  debugging 1
  visualizing performance of 117, 139
parameter 28
partition 1
Partition Manager 10
pdbx Attach screen 8
pdbx debugger 1
  accessing help for dbx subcommands 33
  accessing help for pdbx subcommands 33
  attach mode 6
  checking event status 26
  command context 1
  controlling program execution 20
pdbx debugger *(continued)*
  creating, removing, and listing aliases 34
  deleting breakpoints 25
  deleting events 25
  deleting tracepoints 25
  displaying source 32
  displaying task states 11
  displaying tasks 11
  exiting pdbx 39
  grouping tasks 15
  hooking tasks 26
  interrupting tasks 22
  loading the partition 10
  normal mode 4
  overloaded symbols 38
  reading subcommands from a command file 36
  setting breakpoints 20
  setting command context 15
  setting tracepoints 22
  specifying expressions 36
  specifying variables on trace and stop subcommands 24
  starting pdbx 4
  unhooking tasks 26
  using pdbx 1
  viewing program call stacks 27
  viewing program variables 28
pdbx subcommands 1, 2, 16, 198
  active 11
  alias 34, 198
  assign 199
  attach 200
  attribute 200
  back 201
  call 201
  case 201
  catch 202
  condition 202
  cont 203
  context insensitive subcommands 2
  dbx 203
  delete 25, 204
  detach 39, 205
  dhelp 33, 205
  display memory 206
  down 207
  dump 207
  file 207
  func 208
  goto 208
  gotoi 208
  group 11, 208
  halt 210
  help 33, 211
  hook 26, 211
  ignore 212

pdbx subcommands *(continued)*
list 32, 213
listi 214
load 10, 215
map 215
mutex 215
next 216
nexti 217
on 15, 217
overview 1
print 28, 219
quick reference listing 2
quit 39, 220
registers 220
return 220
search 221
set 221
sh 221
skip 222
source 222
status 26, 222
step 223
stepi 224
stop 20, 224
tasks 226
thread 226
trace 22, 228
unalias 34, 229
unhook 26, 230
unset 231
up 231
use 231
whatis 232
where 27, 232
whereis 232
which 232
PE commands 193
pdbx 193
pedb 233
vt 234
pedb Attach window 44
pedb debugger 41, 65
attach mode 43
changing a variable's format 78
changing a variable's value 78
controlling program execution 57
controlling source code 101
creating task groups 54
customizing pedb resources 107
debugging multiple views 106
debugging programs using multiple views 106
deleting task groups 55
displaying local variables within the program stack 69
displaying variable in more or less detail 76
editing current source file 102
pedb debugger *(continued)*
examining program data 68
exporting array information 81
getting help 106
hiding a task's break/trace information 68
hiding a task's data information 68
hiding a task's stack information 68
leaving pedb 107
loading the partition 48
locating breakpoint in source 68
normal mode 42
setting breakpoints 59
setting the context 52
specifying the array subrange 80
starting pedb 41
stepping execution 63
tracing program execution 65
understanding data types 69
unhooking tasks 67
using pedb 41
viewing the contents of an array 79
visualizing program arrays 97
performance monitoring 139
adjusting sampling interval during 144
selecting nodes for monitoring 143
starting monitoring 143
POE command-line flags
-a 240
-adapter_use 240
-cmdfile 239
-cpu_use 240
-d 240
-euidevelop 239
-euidevice 239
-euilib 239
-euilibpath 239
-hfile 239
-hostfile 239
-I (upper case i) 240
-ilevel 239
-infolevel 239
-labelio 239
-newjob 239
-pgmmodel 240
-pmdlog 239
-pmlights 239
-procs 5, 239
-pulse 240
-resd 239
-retry 239
-retrycount 240
-rmpool 240
-samplefreq 121, 239
-savehostfile 239
-sfreq 121, 239
-spname 240

POE command-line flags *(continued)*
- -stdinmode 239
- -stdoutmode 239
- -tbsize 122, 124, 239
- -tbuffsize 122, 124, 239
- -tbuffwrap 124, 239
- -tbwrap 124, 239
- -tdir 123, 239
- -tfile 121, 239
- -tlevel 120, 239
- -tmpdir 123, 239
- -tracedir 123, 239
- -tracefile 121, 239
- -tracelevel 120, 239
- -ttempsize 122, 124, 239
- -ttsize 122, 124, 239
- -usrport 239
- -vtlibpath 239
- -x 240
- changing the sampling interval for AIX kernel statistics using 121
- specifying a directory for final trace output using 123
- specifying temporary trace file size using 124
- specifying the maximum size of output trace files using 124
- specifying trace buffer size using 124
- specifying trace file names using 121
- specifying wraparound trace storage using 124
- turning tracing on using 120
- writing a trace file to a specified directory using 123

POE environment variables
- changing the sampling interval for AIX kernel statistics using 121
- managing storage for trace files using 122
- MP_DBXPROMPTMOD 197
- MP_DEBUG_INITIAL_STOP 37, 197
- MP_HOSTFILE 236
- MP_INFOLEVEL 121
- MP_PROCS 5
- MP_SAMPLEFREQ 121
- MP_TBUFFSIZE 122, 124
- MP_TBUFFWRAP 124
- MP_TMPDIR 123
- MP_TRACEDIR 123
- MP_TRACEFILE 121
- MP_TRACELEVEL 120
- MP_TTEMPSIZE 122, 124
- specifying a directory for final trace output using 123
- specifying temporary trace file size using 124
- specifying the maximum size of output trace files using 124
- specifying trace buffer size using 124
- specifying trace file names using 121
- specifying wraparound trace storage using 124

POE environment variables *(continued)*
- turning tracing on using 120
- writing a trace file to a specified directory using 123

Point to Point Message Details window 92
pool 240
post-processing 128
procedure 2
processor idle 167
processor utilization 171
processor wait 165
publications, related xii

R

remote node 2, 102
Resource Manager 10
resource settings 247
resources, customizing 247

S

Select Filters window 89
Send/Receive Message Details window 93
server 2
shell script 99
source code 5, 157
source code control 101
source code emphasis 104
source line 21
SPMD (Single Program Multiple Data) 4
standard input (STDIN) 17
standard output (STDOUT) 5
streaming views 115
subcommands 33, 198
- dbx 33, 205
- pdbx 33, 198

subroutine 58
synchronization 118
system calls 188
system summary 190
system views 184

T

task 1
Task Message Queue window 90
task status information 56
Threads Viewer window 74
trace file post-processing 128
trace files
- adjusting playback speed of 135
- cycling playback through 134
- generating 119, 124
- loading for playback 127
- returning playback to an earlier point in 133
- starting playback of 133

trace files *(continued)*
- stepping playback of 134
- using VT to play back 125, 139

trace records 117
- AIX Kernel Statistics 117
- Application Markers 117
- MPI Collective Communication 117
- MPI Message Passing 117

trace visualization 117, 139
- generating trace files for 119, 124
- types of trace records for 117

tracepoint 65
tracing routines 119
trademarks vii

U

user 2
user load balance 173
user utilization 169
user-defined visualizations 99

V

variable 12
Visualization Data Explorer 98, 243
Visualization Tool (VT) 109
- closing views 114
- opening views 114
- performance monitoring 139
- quick operation of 111
- saving screen configurations for 147
- starting 113
- trace visualization 117
- using for performance monitoring 144
- using for trace visualization 139
- view categories described 115
- view descriptions 148, 192

Visualization window 98
VT tracing routines 119
VT views 114, 148
VT views (displays)
- adjusting magnification of 136
- adjusting time resolution for 144
- categories described 115
- changing colors displayed on 144
- closing 114
- Communication/Program views 149
- Computation views 162
- Connectivity Graph 150
- Context Switches 184, 185
  - bar chart 184
  - graph 185
- descriptions of 148
- Disk Reads 174, 175
  - bar chart 174
  - graph 175

VT views (displays) *(continued)*
- Disk Transfers 176, 177
  - bar chart 176
  - graph 177
- Disk views 174
- Disk Writes 178, 179
  - bar chart 178
  - graph 179
- instantaneous views 115
- Interprocessor Communication 151
- Kernel Utilization 162, 164
  - bar chart 162
  - graph 164
- Message Status Matrix 155
- Network views 180
- opening 114
- Packets Received 180, 181
  - bar chart 180
  - graph 181
- Packets Sent 182, 183
  - bar chart 182
  - graph 183
- Page Faults 186, 187
  - bar chart 186
  - graph 187
- Processor Idle 167, 168
  - bar chart 167
  - graph 168
- Processor Utilization 171
  - 3D bar chart 171
- Processor Wait 165, 166
  - bar chart 165
  - graph 166
- scrolling over history buffers 136
- Source Code 157
- streaming views 115
- System Calls 188, 189
  - bar chart 188
  - graph 189
- System Summary 190
- System views 184
- User Load Balance 173
- User Utilization 169, 170
  - bar chart 169
  - graph 170

X

xresources, customizing 247

Communicating Your Comments to IBM

IBM Parallel Environment for AIX
Operation and Use, Volume 2, Part 1
Debugging and Visualizing
Version 2 Release 4

Publication No. SC28-1980-02

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. 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.

If you are mailing a reader's comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - FAX: (International Access Code)+1+914+432-9405
- If you prefer to send comments electronically, use this network ID:
  - IBM Mail Exchange: USIB6TC9 at IBMMAIL
  - Internet e-mail: mhvrcfs@us.ibm.com
  - World Wide Web: http://www.s390.ibm.com/os390

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.

Reader's Comments — We'd Like to Hear from You

IBM Parallel Environment for AIX
Operation and Use, Volume 2, Part 1
Debugging and Visualizing
Version 2 Release 4

Publication No. SC28-1980-02

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: ____________________________________

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

[ ] As an introduction

[ ] As a reference manual

[ ] For another purpose (explain)

[ ] As a text (student)

[ ] As a text (instructor)

____

____

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

Name

Company or Organization

Phone No.

Address

Reader's Comments — We'd Like to Hear from You
SC28-1980-02

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie NY 12601-5400

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Program Number: 5765-543

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC28-1980-02