IBM Parallel Environment for AIX

Operation and Use, Volume 2

Version 3 Release 2

SA22-7426-01

IBM Parallel Environment for AIX

Operation and Use, Volume 2

Version 3 Release 2

SA22-7426-01

Note
Before using this information and the product it supports, read the information in “Notices” on page 231.

Second Edition (December 2001)

This edition applies to version 3, release 2 of IBM Parallel Environment for AIX (product number 5765-D93) and to all subsequent releases and modifications until otherwise indicated in new editions.

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

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

International Business Machines Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+845+432-9405
FAX (Other Countries):
Your International Access Code +1+845+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
Internet e-mail: mhvrcfs@us.ibm.com

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

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

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

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

Contents

Figures

Tables

About this book

This book 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. Specifically, it contains information on PE's debuggers and profiling tools.

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 Guide* , (GC23-3894) and *IBM Parallel Environment for AIX: MPL Programming and Subroutine Reference* (GC23-3893).

This book assumes that AIX 5L™ Version 5.1 (AIX 5L 5.1) 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 @server pSeries or RS/6000® network cluster, or on a mixed system where additional pSeries or 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 5L 5.1 or later documentation listed under "AIX publications" on page 245. 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*.

For a list of related books and details about accessing online information, see "Bibliography" on page 243.

Who should read 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

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. Profiling parallel programs with Xprofiler" on page 33 describes how to profile your programs with the Parallel Environment's Xprofiler.
- "Chapter 3. Analyzing program performance using the PE Benchmarker toolset" on page 97 describes the various tools in the PE Benchmarker toolset. You can use these tools for collecting and analyzing program event trace or hardware performance data.
- "Appendix A. Parallel environment tools commands" on page 139 contains the manual pages for the PE commands discussed throughout this book.

- "Appendix B. Command line flags for normal or attach mode" on page 213 shows the command line flags for **pdbx** debugging in normal or attach mode.
- "Appendix C. Customizing Xprofiler resources" on page 215 describes how to customize X-Windows resources for PE tools.
- "Appendix D. Profiling programs with the AIX prof and gprof commands" on page 223 describes how to use the AIX profilers **prof** and **gprof** to profile parallel programs.
- "Appendix E. Understanding and Creating PCT Hardware Counter Groups" on page 225 describes the pre-defined hardware counter groups we have defined for the Performance Collection Tool (part of the PE Benchmarker toolset). This appendix also describes how you can create your own hardware counter groups.

Conventions and terminology used in this book

This book uses the following typographic conventions:

Convention	Usage
bold	**Bold** words or characters represent system elements that you must use literally, such as: command names, file names, flag names, path names, PE component names (**pdbx**, for example), and subroutines.
italic	*Italicized* 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.

Abbreviated names

Some of the abbreviated names used in this book follow.

Short Name	Full Name
AIX	Advanced Interactive Executive
CSS	communication subsystem
DPCL	dynamic probe class library
dsh	distributed shell
GUI	graphical user interface
HDF	Hierarchical Data Format
IP	Internet Protocol
MPI	Message Passing Interface
MPL	Message Passing Library
PE	IBM Parallel Environment for AIX
PE MPI	IBM's implementation of the MPI standard for PE
PE MPI-IO	IBM's implementation of MPI I/O for PE
PM array	program marker array
POE	parallel operating environment
pSeries	IBM @server pSeries
PSSP	IBM Parallel System Support Programs for AIX
RISC	reduced instruction set computer

Short Name	Full Name
rsh	remote shell
RS/6000	IBM RS/6000
SDR	System Data Repository
SP	IBM RS/6000 SP
STDERR	standard error
STDIN	standard input
STDOUT	standard output
US	user space

How to send your comments

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

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

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

National language support

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 PE, 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 these 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*, GA22-7419 and *IBM AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs*.

What's new in Parallel Environment 3.2?

This release of the Parallel Environment contains a number of functional enhancements, including:

- new PE Benchmarker tools
- improved parallel checkpointing capabilities
- support for 64-bit applications
- MPI enhancements

In addition, PE 3.2 includes these changes:

- PE now supports, and requires, AIX 5L 5.1.
- PE Version 2, Release 2 is no longer supported.
- The Dynamic Probe Class Library (DPCL) is no longer a part of PE, though it is still shipped with PE. Instead, DPCL is now an open source offering that supports PE.
- The **pedb** debugger has been removed.
- The VT parallel tracing facility has been removed.

The following sections describe these functional enhancements and changes in more detail.

New PE Benchmarker tools

This release of PE contains a new suite of applications and utilities that you can use to analyze the performance of programs. This suite of tools is called *PE Benchmarker* and contains:

- the *Performance Collection Tool*, which enables you to collect either MPI and user event data, or else hardware and operating system profiles for one or more application processes. This tool is built on dynamic instrumentation technology (the Dynamic Probe Class Library, or DPCL) which enables you to make the decision of what data to collect at run time. Probes are placed in the running executable to collect just the information you requested. This ability typically results in a more acceptable intrusion cost than you would have with more traditional styles of instrumentation.
- a set of Unified Trace Environment (UTE) utilities for converting one or more AIX trace files (as output by the Performance Collection Tool) into UTE interval files. Additional UTE utilities enable you to:
  - generate statistics tables from UTE interval files.
  - convert UTE interval files into a single SLOG file. You can analyze an SLOG file using Jumpshot (a public domain tool developed by Argonne National Laboratory).
- the *Profile Visualization Tool*, for viewing hardware and operating system profiles collected by the Performance Collection Tool.

Improved parallel checkpointing capabilities

This release of PE includes more flexible parallel checkpoint/restart capabilities. In previous releases, only POE/MPI applications submitted under LoadLeveler® in batch mode could be checkpointed, and there were significant limitations. What's more, a checkpoint sequence could be initiated only by all tasks in the parallel MPI program. In this release, PE's checkpointing capabilities have been extended to allow:

- a user to initiate a checkpoint sequence explicitly by issuing the new command **poeckpt**. Another new command, **poerestart**, enables a user to restart a POE job using a checkpoint file.
- a single task in a parallel MPI or LAPI job to initiate a checkpoint sequence. It is no longer necessary for all tasks in the job to call checkpointing functions.
- a system administrator or LoadLeveler to initiate a checkpoint sequence.

Also, many significant limitations of the checkpointing capabilities have been removed, and the checkpointing compiler scripts (**mpcc_chkpt**, for example) are no longer used.

MPI enhancements

With this release, PE MPI provides all of the functions in the MPI standard, except for the functionality defined in the ″Process Creation and Management″ chapter of MPI-2.

Additional command for starting MPI jobs

While you can continue to use the **poe** command to start MPI jobs, this release of PE provides support for the **mpiexec** command described in the MPI-2 standard. This command is not meant to replace the **poe** command; instead it is provided as a portable way to start MPI programs, and should prove helpful for applications that target multiple implementations of MPI.

Support for 64–bit applications

Certain architectures that PE supports, such as POWER3 SMP High Node, POWER3 SMP Thin Node, and POWER3 Wide Node, can run applications using a 64-bit address space. Accordingly, the tools that are provided with PE, as well as the MPI threads library, have been enhanced to support 64-bit applications on 64-bit processors.

MPI-IO performance enhancements

To improve the performance of IBM's implementation of MPI-IO, a variety of optimizations have been made to this release of PE. For example, two new POE environment variable/command-line flag pairs and one new file hint have been added. The two new POE environment variable/command-line flag pairs are the:

- **MP_IO_BUFFER_SIZE** environment variable (**-io_buffer_size** flag), which enables you to set the default size of buffers used by I/O agents.
- **MP_IO_ERRLOG** environment variable (**-io_errlog** flag), which enables you to log errors that occurred at the file system level throughout the MPI-IO application run.

The new file hint, **IBM_sparse_access**, enables you to specify whether the file access requests from participating tasks are sparse or dense.

Extended collective communication

This release provides the MPI_IN_PLACE and intercommunicator semantic that MPI-2 has added to a number of MPI 1.1 collective communication subroutines. The new subroutines, MPI_ALLTOALLW and MPI_EXSCAN, are also provided. The nonblocking collective subroutines, which have a prefix of MPE_I, are not enhanced.

C++ and FORTRAN90 support

This release provides the C++ bindings described by MPI-2. C++ programmers can now use PE MPI more naturally, as they no longer need to use the C bindings.

FORTRAN programs can now use the **mpi** module in place of **mpif.h**.

The nonblocking collective subroutines, which have a prefix of MPE_I, are not enhanced for C++.

MPI-2 external interfaces support

With this release, PE now provides full support of MPI-2 external interfaces in the MPI threads library. This support enables you to layer additional functionality on top of MPI with an interface that is similar to MPI's.

Miscellaneous MPI-2 enhancements

Some of the functions included in the ″Miscellany″ chapter of MPI-2 were provided in prior releases of PE. This release provides the rest of these miscellaneous functions.

DPCL is now an open source offering

The Dynamic Probe Class Library (DPCL) is no longer a part of the IBM PE for AIX licensed program, but it is still shipped with PE for convenience. Instead, DPCL is now available as an open source offering that supports PE. For more information on the DPCL open source project, go to this World Wide Web address:

http://oss.software.ibm.com/developerworks/opensource/dpcl/

Removal of pedb debugger support

Beginning with this release, PE no longer includes the **pedb** parallel debugger. As a result, the **pedb** command is no longer available. The **pdbx** parallel debugger, however, is still supported; use it instead of **pedb** to debug your parallel applications.

Removal of VT trace collection support

Beginning with this release, PE no longer supports the parallel trace collection facility formerly used by the PE Visualization Tool (which was removed from PE in release 3.1). Because the usefulness of examining these traces independent of VT is limited, the VT trace facility has been replaced by the more robust tracing capabilities of the new PE Benchmarker suite of tools.

Commands no longer supported

Beginning with this release, PE no longer supports these commands:

- **mpcc_chkpt**
- **mpCC_chkpt**
- **mpxlf_chkpt**
- **mpxlf90_chkpt**
- **mpxlf95_chkpt**
- **pedb**

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 *AIX 5L Version 5.1 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 pSeries or 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 and Table 2 on page 3 outline 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 139.

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 14.

Table 1. 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 28
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 32
dhelp [dbx_command]	Display a brief list of **dbx** commands or help information about them.	“Accessing help for dbx subcommands” on page 27
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 27
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 14
quit	End a **pdbx** session.	“Exiting pdbx” on page 32
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 29
tasks [long]	Display information about all the tasks in the partition.	“Displaying tasks and their states” on page 10
unalias alias_name	Remove a command alias specified by the **alias** subcommand.	“Creating, removing, and listing command aliases” on page 28

Table 2. 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 22
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 139.
hook	Regain control over an unhooked task.	"Unhooking and hooking tasks" on page 23
list [line_number | line_number, line_number | procedure]	Display lines of the current source file, or of a procedure.	"Displaying source" on page 26
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 9
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 24
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 23
stop	Set a breakpoint for tasks in the current context. Breakpoints are stopping places in your program that halt execution.	"Setting breakpoints" on page 18
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 20
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 23
where	Display a list of active procedures and functions.	"Viewing program call stacks" on page 24
<**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 16

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 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 14.

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 3. 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 3. 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 29 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 3. Debugger Option Flags (pdbx) (continued)

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 *AIX 5L Version 5.1 Commands Reference* and *AIX 5L Version 5.1 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 139.

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.

```
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 11 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 213 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 4 on page 14 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:**	**Then:**

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 4. 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.

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”.

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 16 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 28
back	Return to a **pdbx** prompt.	"Returning to a pdbx prompt" on page 17

This subset command:	Is used to:	For more information see:
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 19
help [subject]	Display a list of **pdbx** commands and topics or help information about them.	"Accessing help for pdbx subcommands" on page 27
on <[group | task]>	Set the current context for later subcommands. This command always returns to a **pdbx** prompt.	"Setting command context" on page 14
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 29
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 23
tasks [long]	Display processes (tasks) and their states.	"Displaying tasks and their states" on page 10
quit	Exit the **pdbx** program and kill the application.	"Exiting pdbx" on page 32
unalias alias_name	Remove a previously defined alias.	"Creating, removing, and listing command aliases" on page 28
<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 16

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 *AIX 5L Version 5.1 Commands Reference* and *AIX 5L Version 5.1 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 29.

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 21 for more information.
3. For further details on the **stop** subcommand, refer to its use on the **dbx** command as described in *AIX 5L Version 5.1 Commands Reference* and *AIX 5L Version 5.1 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 29.

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” for more information.
3. For further detail on the **trace** subcommand, refer to its use on the **dbx** command as described in *AIX 5L Version 5.1 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 29 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 *AIX 5L Version 5.1 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 3 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 139 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 may 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. Profiling parallel programs with Xprofiler

This chapter describes how to profile your programs with the Xprofiler profiling tool of the IBM Parallel Environment for AIX. This chapter explains how to use the Xprofiler graphical user interface (GUI) to profile your application, so it is best to read it while you have the GUI up and running.

If you intend to use the AIX **gprof** command to profile your parallel application, see "Appendix D. Profiling programs with the AIX prof and gprof commands" on page 223 for information on how to do so. You may also find it helpful to consult the *AIX 5L Version 5.1 Commands Reference*.

You do not need to be familiar with the AIX **gprof** command to use Xprofiler.

Xprofiler is a tool that helps you analyze your parallel application's performance quickly and easily. It uses data collected by the **-pg** compiling option to construct a graphical display of the functions within your application. Xprofiler provides quick access to the profiled data, which lets you identify the functions that are the most CPU-intensive. The graphical user interface also lets you manipulate the display in order to focus on the application's critical areas.

Before you begin

About Xprofiler

Xprofiler lets you profile both serial and parallel applications. The difference is that when you run a serial application, a single profile data file is generated, while a parallel application produces multiple profile data files. Either way, you can use Xprofiler to analyze the resulting profiling information.

Xprofiler provides a set of resource variables that let you customize some of the features of the Xprofiler window and reports. For information about customizing resources for Xprofiler, see *IBM Parallel Environment for AIX: Operation and Use, Volume 2, Tools Reference*

The word *function* is used frequently throughout this chapter. Consider it to be synonymous with the terms *routine*, *subroutine*, and *procedure*.

Requirements and limitations

To use Xprofiler, your application must be compiled with the **-pg** option. For more information about compiling, see "Compiling applications to be profiled" on page 34.

Like gprof, Xprofiler lets you analyze CPU (busy) usage only. It cannot give you other kinds of information such as CPU idle, I/O, or communication.

If you compile your application on one machine, and then analyze it on another, you must first make sure that both machines have similar library configurations, at least for the system libraries used by the application. For instance, say you ran an HPF application on an SP, then tried to analyze the profiled data on a workstation. The levels of HPF runtime libraries must match, and must be placed in a location that Xprofiler recognizes on the workstation. Otherwise, Xprofiler produces unpredictable results.

Since Xprofiler collects data by sampling, short-executing functions may show no CPU use.

Xprofiler does not give you information about the specific threads in a multi-threaded program. The data that Xprofiler presents is a summary of the activities of all the threads.

Xprofiler versus gprof

With Xprofiler, you can produce the same tabular reports that you may be accustomed to seeing with **gprof**. Just as with **gprof**, you can generate the Flat Profile, Call Graph Profile, and Function Index reports. Xprofiler is different from **gprof** in that it provides a graphical user interface (GUI) from which you can profile your application. It generates a graphical display of your application's performance, as opposed to just a text-based report. Unlike **gprof**, Xprofiler also lets you profile your application at the source statement level.

From the Xprofiler GUI, you can use all the same command line flags as **gprof**, plus a few more that are unique to Xprofiler.

Compiling applications to be profiled

In order to use Xprofiler, you must compile and link your application with the **-pg** option of the compiler command. This applies regardless of whether you are compiling a serial or parallel application. You can compile and link your application all at once, or perform the compile and link operations separately. Here's an example of how you would compile and link all at once:

```
cc -pg -o foo foo.c
```

And here's an example of how you would first compile your application and then link it. To compile:

```
cc -pg -c foo.c
```

To link:

```
cc -pg -o foo foo.o
```

Notice that when you compile and link separately, you must use the -pg option with *both* the compile and link commands.

The **-pg** option compiles and links the application so that when you run it, the CPU usage data gets written to one or more output files. For a serial application, this output consists of just one file called *gmon.out*, by default. For parallel applications, the output is written into multiple files, one for each task that is running in the application. To prevent each output file from overwriting the others, POE appends the task ID to each gmon.out file. For instance, *gmon.out.10*.

The **-pg** option is not a combination of the **-p** and the **-g** compiling options.

In order to get a complete picture of your parallel application's performance, you must indicate all of its gmon.out files when you load the application into Xprofiler. When you specify more than one gmon.out file, Xprofiler shows you the sum of the profile information contained in each file.

The Xprofiler GUI provides the capability of viewing included functions. Note, however, that your application must also be compiled with the **-g** option in order for Xprofiler to display the included functions.

The **-g** option, in addition to the **-pg** option, is also required for source statement profiling.

Starting Xprofiler

You start Xprofiler by issuing the **xprofiler** command from the AIX command line. You also need to specify the executable, profile data file(s), and options, which you can do one of two ways. You can either specify them on the command line, with the **xprofiler** command, or you can issue the **xprofiler** command alone, then specify them from within the GUI. See “Loading files from the Xprofiler GUI” on page 38.

Under some circumstances you may have multiple profile data files (gmon.out files). You will have more than one of these files if you are profiling a parallel application, because a gmon.out file is created for each task in the application when it is run. If you are running a serial application, there may be times when you want to summarize the profiling results from multiple runs of the application. In these cases, you will need to specify each of the profile data files you want to profile with Xprofiler.

To start Xprofiler and specify the executable, profile data file(s), and options:

ENTER **xprofiler** *a.out gmon.out...* [*options*]

where *a.out* is the binary executable file, *gmon.out* is the name of your profile data file(s), and *options* may be one or more of the options listed in “Xprofiler command line options”.

To print basic Xprofiler command syntax to the screen, use the *-h* or *-?* option with the **xprofiler** command from the command line. For example, **xprofiler** *-h*.

Xprofiler command line options

You can specify the same command line options with the **xprofiler** command that you do with **gprof**, plus one additional option (**-disp_max**), which is specific to Xprofiler. The command line options let you control the way Xprofiler displays the profiled output.

When you enter an option, there must be a space between the option and its corresponding value. For example,

```
-e main
```

You can specify the following options from either the Xprofiler GUI or the command line. See “Specifying command line options (from the GUI)” on page 43 for more information.

The Xprofiler command line options are:

Table 5. Xprofiler Command Line Options

Use this flag:	To:	For example:
-b	Suppresses the printing of the field descriptions for the Flat Profile, Call Graph Profile, and Function Index reports when they are written to a file with the **Save As** option of the File menu.	To suppress printing of the field descriptions for the Flat Profile, Call Graph Profile, and Function Index reports in the saved file: file, **ENTER** **xprofiler -b** *a.out gmon.out*

Table 5. Xprofiler Command Line Options (continued)

Use this flag:	To:	For example:
-s	If multiple gmon.out files are specified when Xprofiler is started, produces the gmon.sum profile data file. The gmon.sum file represents the sum of the profile information in all the specified profile files. Note that if you specify a single gmon.out file, the gmon.sum file contains the same data as the gmon.out file.	To write the sum of the data from three profile data files, *gmon.out.1*, *gmon.out.2*, and *gmon.out.3*, into a file called *gmon.sum*: **ENTER** **xprofiler -s** *a.out gmon.out.1 gmon.out.2 gmon.out.3*
-z	Includes functions that have both zero CPU usage and no call counts in the Flat Profile, Call Graph Profile, and Function Index reports. A function will not have a call count if the file that contains its definition was not compiled with the **-pg** option, which is common with system library files.	To include all functions used by the application, in the Flat Profile, Call Graph Profile, and Function Index reports, that have zero CPU usage and no call counts: **ENTER** **xprofiler -z** *a.out gmon.out*
-a	Adds alternative paths to search for source code and library files, or changes the current path search order. When using this command line option, you can use the "at" symbol (@) to represent the default file path, in order to specify that other paths be searched before the default path.	To set the alternative file search path(s) so that Xprofiler searches *pathA*, the default path, then *pathB*: **ENTER** **xprofiler -a** *pathA:@:pathB*
-c	Loads the specified configuration file. If the **-c** option is used on the command line, the configuration file name specified with it will appear in the **Configuration File (-c):** text field in the *Load Files Dialog*, and the **Selection** field of the *Load Configuration File Dialog*. When both the **-c** and **-disp_max** options are specified on the command line, the **-disp_max** option is ignored, but the value that was specified with it will appear in the **Initial Display (-disp_max):** field in the Load Files Dialog, the next time it is opened.	To load the configuration file: **ENTER** **xprofiler** *a.out gmon.out* **-c** *config_file_name*
-disp_max	Sets the number of function boxes that Xprofiler initially displays in the function call tree. The value supplied with this flag can be any integer between 0 and 5,000. Xprofiler displays the function boxes for the most CPU-intensive functions through the number you specify. For instance, if you specify 50, Xprofiler displays the function boxes for the 50 functions in your program that consume the most CPU. After this, you can change the number of function boxes that are displayed via the Filter menu options. This flag has no effect on the content of any of the Xprofiler reports.	To display the function boxes for only 50 most CPU-intensive functions in the function call tree: **ENTER** **xprofiler -disp_max** *50 a.out gmon.out*

Table 5. Xprofiler Command Line Options (continued)

Use this flag:	To:	For example:
-e	De-emphasizes the general appearance of the function box(es) for the specified function(s) in the function call tree, and limits the number of entries for these function in the Call Graph Profile report. This also applies to the specified function's descendants, as long as they have not been called by non-specified functions. In the function call tree, the function box(es) for the specified function(s) appears greyed-out. Its size and the content of the label remain the same. This also applies to descendant functions, as long as they have not been called by non-specified functions. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. The information for this entry remains unchanged. Entries for descendants of the specified function do not appear unless they have been called by at least one non-specified function in the program.	To deemphasize the appearance of the function boxes for *foo* and *bar*, as well as their qualifying descendants in the function call tree, and limit their entries in the Call Graph Profile report: **ENTER** **xprofiler -e** *foo* **-e** *bar a.out gmon.out*
-E	Changes the general appearance and label information of the function box(es) for the specified function(s) in the function call tree. Also limits the number of entries for these functions in the Call Graph Profile report, and changes the CPU data associated with them. These results also apply to the specified function's descendants, as long as they have not been called by non-specified functions in the program. In the function call tree, the function box for the specified function appears greyed-out, and its size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0 (zero). The same applies to function boxes for descendant functions, as long as they have not been called by non-specified functions. This option also causes the CPU time spent by the specified function to be deducted from the left side CPU total in the label of the function box for each of the specified function's ancestors. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. When this is the case, the time in the *self* and *descendants* columns for this entry is set to 0 (zero). In addition, the amount of time that was in the *descendants* column for the specified function is subtracted from the time listed under the *descendants* column for the profiled function. As a result, be aware that the value listed in the *% time* column for most profiled functions in this report will change.	To change the display and label information for *foo* and *bar*, as well as their qualifying descendants in the function call tree, and limit their entries and data in the Call Graph Profile report: **ENTER** **xprofiler -E** *foo* **-E** *bar a.out gmon.out*

Table 5. Xprofiler Command Line Options (continued)

Use this flag:	To:	For example:
-f	De-emphasizes the general appearance of all function boxes in the function call tree, *except* for that of the specified function(s) and its descendant(s). In addition, the number of entries in the Call Graph Profile report for the non-specified functions and non-descendant functions is limited. The **-f** flag overrides the **-e** flag. In the function call tree, all function boxes *except* for that of the specified function(s) and its descendant(s) appear greyed-out. The size of these boxes and the content of their labels remain the same. For the specified function(s), and its descendant(s), the appearance of the function boxes and labels remain the same. In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. All information for this entry remains the same.	To de-emphasize the display of function boxes for all functions in the function call tree *except* for *foo*, *bar*, and their descendants, and limit their types if entries in the Call Graph Profile report: **ENTER** **xprofiler -f** *foo* **-f** *bar a.out gmon.out*
-F	Changes the general appearance and label information of all function boxes in the function call tree *except* for that of the specified function(s) and its descendants. In addition, the number of entries in the Call Graph Profile report for the non-specified and non-descendant functions is limited, and the CPU data associated with them is changed. The **-F** flag overrides the **-E** flag. In the function call tree, all function boxes *except* for that of the specified function(s) and its descendant(s) appear greyed-out. The size and shape of these boxes also changes so that they appear as squares of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0 (zero). In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. When this is the case, the time in the *self* and *descendants* columns for this entry is set to 0 (zero). As a result, be aware that the value listed in the *% time* column for most profiled functions in this report will change.	To change the display and label information of the function boxes for all functions *except* the functions *foo* and *bar* and their descendants, and limit their types of entries and data in the Call Graph Profile: **ENTER** **xprofiler -F** *foo* **-F** *bar a.out gmon.out*
-L	Sets the pathname for locating shared libraries. If you plan to specify multiple paths, use the *Set File Search Path* option of the File menu on the Xprofiler GUI. See “Setting the file search sequence” on page 46 for information.	To specify */lib/profiled/libc.a:shr.o* as an alternate pathname for your shared libraries: **ENTER** **xprofiler -L** */lib/profiled/libc.a:shr.o*

After you issue the **xprofiler** command, the Xprofiler main window appears, and displays your application’s data.

Loading files from the Xprofiler GUI

If you issue the **xprofiler** command without specifying an executable file, a profile data file, or options, you may do so from within the Xprofiler GUI. You use the *Load File* option of the File menu to do this.

When you issue the **xprofiler** command alone, the Xprofiler main window appears. Since you did not load an executable or specify a profile data file, the window will be empty.

If you issue the **xprofiler** command with the **-h** option only, Xprofiler displays the syntax for the command and then exits.

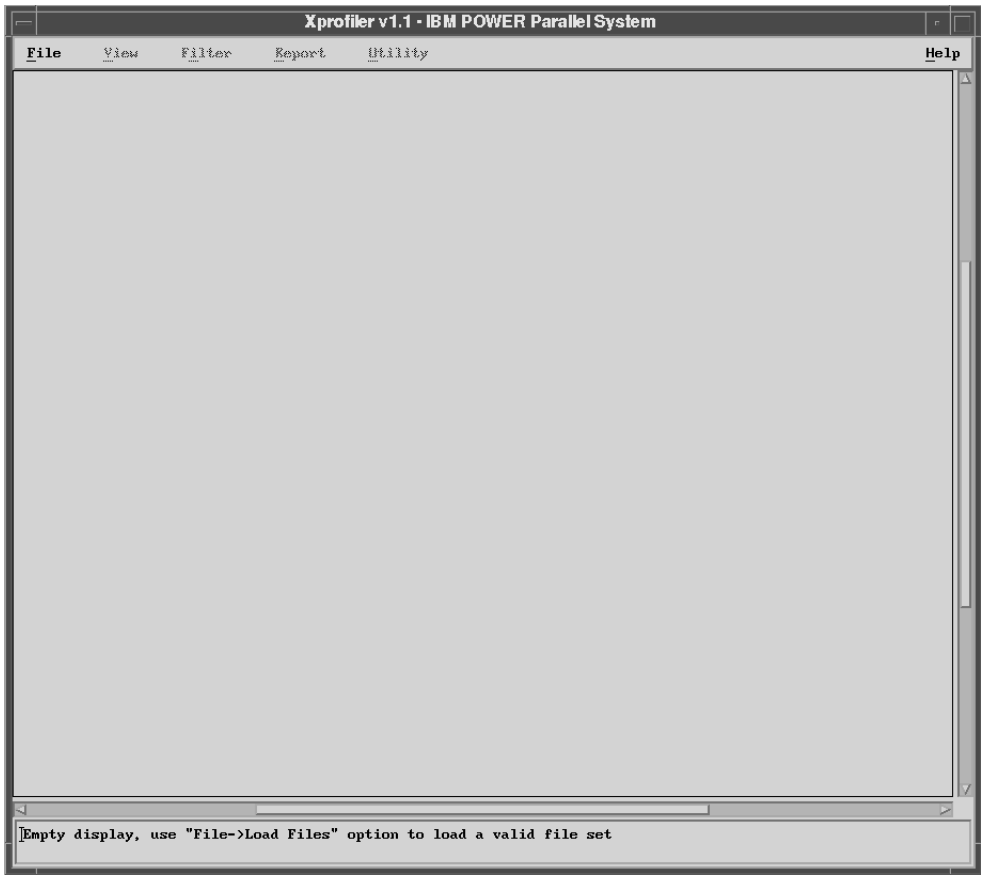


Figure 2. Xprofiler Main Window with No Executables Loaded

Select the File menu, and then the *Load File* option. The *Load Files Dialog* window appears.

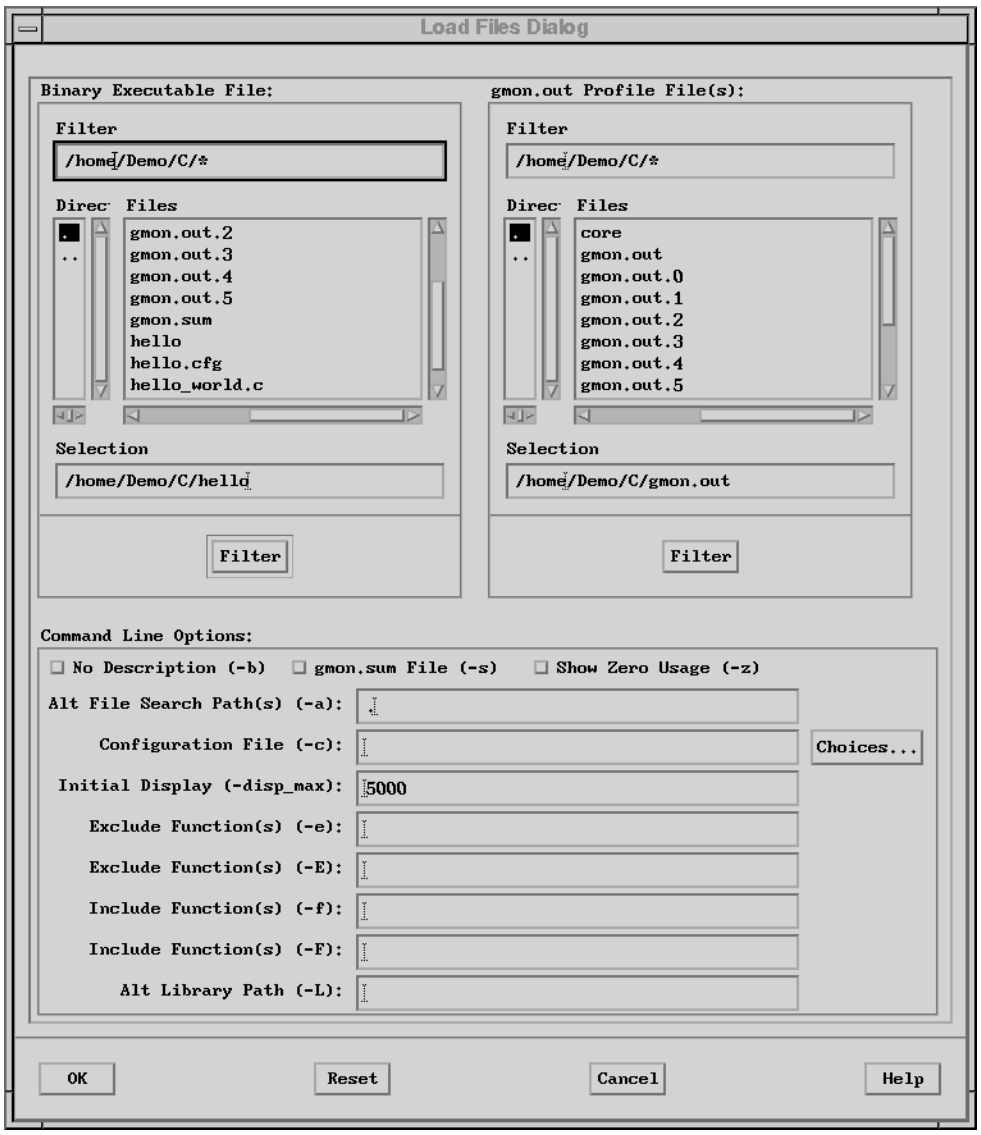


Figure 3. Load Files Dialog Window

The Load Files Dialog window lets you specify your application's executable, and its corresponding profile data (gmon.out) files. When you load a file, you can also specify the various command line options that let you control the way Xprofiler displays the profiled data.

To load the files for the application you wish to profile, you need to specify the:

- *Binary Executable* (required)
- *Profile Data File(s)* (required)
- *Command Line Options* (optional)

Specifying the binary executable

You specify the binary executable from the *Binary Executable File* of the *Load Files Dialog* window. The Binary Executable File area looks similar to this:

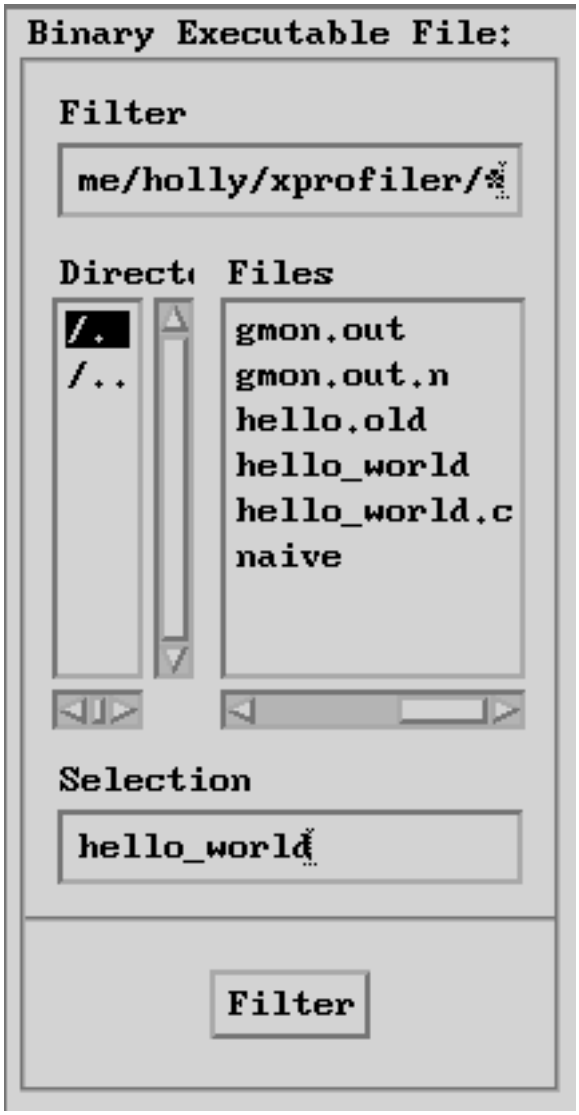


Figure 4. Binary Executable File Area

Use the scroll bars of the *Directories* and *Files* selection boxes to locate the executable file you wish to load. By default, all the files in the directory from which you invoked Xprofiler appear in the Files selection box. To select a file, click on it with the left mouse button.

To make locating your binary executable files easier, the Binary Executable File area includes a *Filter* button. Filtering lets you limit the files that are displayed in the *Files* selection box to those of a specific directory or of a specific type. For information on using the Filter, see “Using the dialog window filters” on page 54.

Specifying the profile data file(s)

You specify the profile data file(s) from the *gmon.out Profile Data File(s)* area of the *Load Files Dialog* window. The gmon.out Profile Data File(s) area looks similar to this:

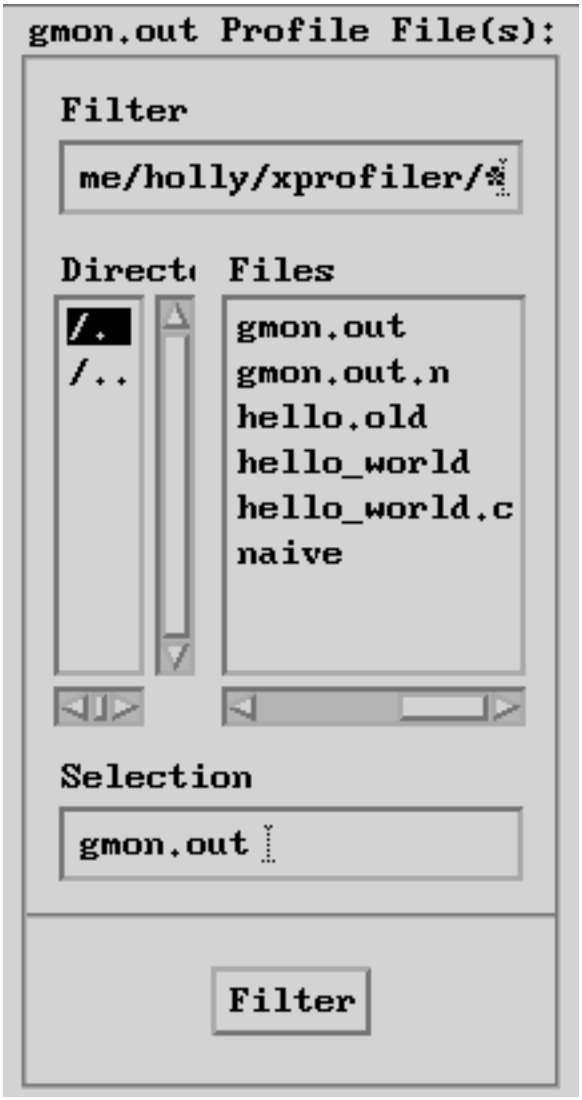


Figure 5. gmon.out Profile Data File(s) Area

When you started Xprofiler, with the **xprofiler** command, you were not required to indicate the name of the profile data file (which is probably why you are specifying it from the GUI). If you did not specify a profile data file, Xprofiler searches your directory for the presence of a file named *gmon.out* and, if found, places it in the *Selection* field of the gmon.out Profile File(s) area, as the default. Xprofiler then uses this file as input, even if it is not related to the binary executable file you specify. Since this will cause Xprofiler to display incorrect data, it is important that you enter the correct file into this field. So, if the profile data file you wish to use is named something other than what appears in the Selection field, you must replace it with the correct file name, as follows.

Use the scroll bars of the *Directories* and *Files* selection boxes to locate one or more of the profile data (gmon.out) files you wish to specify. The file you use does not have to be named *gmon.out*, and you may specify more than one profile data file. To select a file, click on it with the left mouse button. You can select multiple files by holding down the <**Ctrl**> key and clicking on each one with the left mouse button. To select multiple consecutive files, press and hold the left mouse button

over the first file, and then drag the mouse over the other files. To de-select a file, press and hold the <**Ctrl**> key while clicking on the file.

To make locating your output files easier, the gmon.out Profile File(s) area includes a *Filter* button. Filtering lets you limit the files that get displayed in the Files selection box to those in a specific directory or of a specific type. For information on using the Filter, see “Using the dialog window filters” on page 54.

Specifying command line options (from the GUI)

You specify command line options from the *Command Line Options* area of the *Load Files Dialog* window. The Command Line Options area looks similar to this:

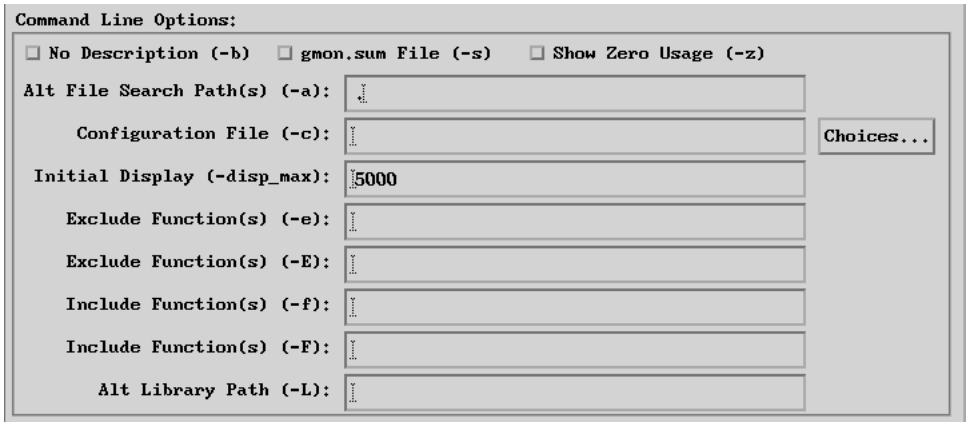


Figure 6. Command Line Options Area

You may specify one or more options as follows:

Table 6. Xprofiler GUI Command Line Options

Use this flag:	To:	For example:
-b (button)	Suppresses the printing of the field descriptions for the Flat Profile, Call Graph Profile, and Function Index reports when they are written to a file with the **Save As** option of the File menu.	To suppress printing of the field descriptions for the Flat Profile, Call Graph Profile, and Function Index reports in the saved file, set the *-b* button to the pressed-in position.
-s (button)	If multiple gmon.out files are specified in the *gmon.out Profile File(s)* area, produces the gmon.sum profile data file. The gmon.sum file represents the sum of the profile information in all the specified profile files. Note that if you specify a single gmon.out file, the gmon.sum file contains the same data as the gmon.out file.	To write the sum of the data from three profile data files, *gmon.out.1*, *gmon.out.2*, and *gmon.out.3*, into a file called *gmon.sum*, set the *-s* button to the pressed-in position to activate this option.
-z (button)	Includes functions that have both zero CPU usage and no call counts in the Flat Profile, Call Graph Profile, and Function Index reports. A function will not have a call count if the file that contains its definition was not compiled with the **-pg** option, which is common with system library files.	To include all functions used by the application, in the Flat Profile, Call Graph Profile, and Function Index reports, that have zero CPU usage and no call counts, set the *-z* button to the pressed-in position to activate this option.

Table 6. Xprofiler GUI Command Line Options (continued)

Use this flag:	To:	For example:
-a (field)	Adds alternative paths to search for source code and library files, or changes the current path search order. After clicking on the **OK** button, any modifications to this field are also made to the **Enter Alt File Search Paths:** field of the *Alt File Search Path Dialog* window. If both the *Load Files Dialog* window and the Alt File Search Path Dialog window are opened at the same time, when you make path changes in the Alt File Search Path Dialog and click the **OK** button, these changes are also made to the Load Files Dialog window. Also, when both of these windows are open concurrently, clicking on the **OK** or **Cancel** buttons in the Load Files Dialog causes both windows to close. If you wish to restore the **Alt File Search Path(s) (-a):** field to the same state as when the Load Files Dialog window was opened, click on the **Reset** button. You can use the "at" symbol (@) with this option to represent the default file path, in order to specify that other paths be searched before the default path.	To set the alternative file search path(s) so that Xprofiler searches *pathA*, the default path, then *pathB*, type *pathA:@:pathB* in the *Alt File Search Path(s) (-a)* field.
-c (field)	Loads the specified configuration file. If the **-c** option was used on the command line, or a configuration file had been previously loaded with the *Load Files Dialog* or *Load Configuration File Dialog* windows, the name of the most recently loaded file will appear in the **Configuration File (-c):** text field in the Load Files Dialog, as well as the **Selection** field of the Load Configuration File Dialog. If both the Load Files Dialog and Load Configuration File Dialog windows are open at the same time, when you specify a configuration file in the Load Configuration File Dialog and then click the **OK** button, the name of the specified file also appears in the Load Files Dialog. Also, when both of these windows are open concurrently, clicking on the **OK** or **Cancel** buttons in the Load Files Dialog causes both windows to close. When entries are made to both the **Configuration File (-c):** and **Initial Display (-disp_max):** fields in the Load Files Dialog, the value in the **Initial Display (-disp_max):** field is ignored, but is retained the next time this window is opened. If you wish to retrieve the file name that was in the **Configuration File (-c):** field when the Load Files Dialog window was opened, click on the **Reset** button.	To load the configuration file, type *gmon.out* in the *Configuration File (-c)* field.
-disp_max (field)	Sets the number of function boxes that Xprofiler initially displays in the function call tree. The value supplied with this flag can be any integer between 0 and 5,000. Xprofiler displays the function boxes for the most CPU-intensive functions through the number you specify. For instance, if you specify 50, Xprofiler displays the function boxes for the 50 functions in your program that consume the most CPU. After this, you can change the number of function boxes that are displayed via the Filter menu options. This flag has no effect on the content of any of the Xprofiler reports.	To display the function boxes for only 50 most CPU-intensive functions in the function call tree, type 50 in the *Init Display (-disp_max) field*

Table 6. Xprofiler GUI Command Line Options (continued)

Use this flag:	To:	For example:
-e (field)	De-emphasizes the general appearance of the function box(es) for the specified function(s) in the function call tree, and limits the number of entries for these function in the Call Graph Profile report. This also applies to the specified function's descendants, as long as they have not been called by non-specified functions. In the function call tree, the function box(es) for the specified function(s) appears greyed-out. Its size and the content of the label remain the same. This also applies to descendant functions, as long as they have not been called by non-specified functions. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. The information for this entry remains unchanged. Entries for descendants of the specified function do not appear unless they have been called by at least one non-specified function in the program.	To de-emphasize the appearance of the function boxes for *foo* and *bar*, as well as their qualifying descendants in the function call tree, and limit their entries in the Call Graph Profile report, type *foo* and *bar* in the *Exclude Routines (-e)* field. You specify multiple functions by separating each one with a space.
-E (field)	Changes the general appearance and label information of the function box(es) for the specified function(s) in the function call tree. Also limits the number of entries for these functions in the Call Graph Profile report, and changes the CPU data associated with them. These results also apply to the specified function's descendants, as long as they have not been called by non-specified functions in the program. In the function call tree, the function box for the specified function appears greyed-out, and its size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0 (zero). The same applies to function boxes for descendant functions, as long as they have not been called by non-specified functions. This option also causes the CPU time spent by the specified function to be deducted from the left side CPU total in the label of the function box for each of the specified function's ancestors. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. When this is the case, the time in the *self* and *descendants* columns for this entry is set to 0 (zero). In addition, the amount of time that was in the *descendants* column for the specified function is subtracted from the time listed under the *descendants* column for the profiled function. As a result, be aware that the value listed in the *% time* column for most profiled functions in this report will change.	To change the display and label information for *foo* and *bar*, as well as their qualifying descendants in the function call tree, and limit their entries and data in the Call Graph Profile report, type *foo* and *bar* in the *Exclude Routines (-E)* field. You specify multiple functions by separating each one with a space.

Table 6. Xprofiler GUI Command Line Options (continued)

Use this flag:	To:	For example:
-f (field)	De-emphasizes the general appearance of all function boxes in the function call tree, *except* for that of the specified function(s) and its descendant(s). In addition, the number of entries in the Call Graph Profile report for the non-specified functions and non-descendant functions is limited. The **-f** flag overrides the **-e** flag. In the function call tree, all function boxes *except* for that of the specified function(s) and its descendant(s) appear greyed-out. The size of these boxes and the content of their labels remain the same. For the specified function(s), and its descendants, the appearance of the function boxes and labels remain the same. In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. All information for this entry remains the same.	To de-emphasize the display of function boxes for all functions in the function call tree *except* for *foo*, *bar*, and their descendants, and limit their types if entries in the Call Graph Profile report, type *foo* and *bar* in the *Include Routines (-f)* field. You specify multiple functions by separating each one with a space.
-F (field)	Changes the general appearance and label information of all function boxes in the function call tree *except* for that of the specified function(s) and its descendants. In addition, the number of entries in the Call Graph Profile report for the non-specified and non-descendant functions is limited, and the CPU data associated with them is changed. The **-F** flag overrides the **-E** flag. In the function call tree, all function boxes *except* for that of the specified function(s) and its descendant(s) appear greyed-out. The size and shape of these boxes changes so that they appear as squares of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0 (zero). In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. When this is the case, the time in the *self* and *descendants* columns for this entry is set to 0 (zero). As a result, be aware that the value listed in the *% time* column for most profiled functions in this report will change.	To change the display and label information of the function boxes for all functions *except* the functions *foo* and *bar* and their descendants, and limit their types of entries and data in the Call Graph Profile, type *foo* and *bar* in the *Include Routines (-F)* field. You specify multiple functions by separating each one with a space.
-L (field)	Sets the alternate pathname for locating shared objects. If you plan to specify multiple paths, use the *Set File Search Path* option of the File menu on the Xprofiler GUI. See “Setting the file search sequence” for information.	Type the alternate library pathname in this field.

Once you have specified the binary executable, the profile data file, and any command line options you wish to use, press the **OK** button to save the changes and close the dialog window. Xprofiler loads your application and displays its performance data.

Setting the file search sequence

You can specify where you want Xprofiler to look for your library files and source code files by using the *Set File Search Paths* option of the File menu. By default, Xprofiler searches the default paths first and then any alternative paths you specify.

Default paths

For library files, Xprofiler uses the paths recorded in the specified gmon.out file(s). If you use the -L command line option, the path you specify with this option will be used instead of those in the gmon.out file.

Note: -L allows only one path to be specified and you can use this option only once.

For source code files, the paths recorded in the specified a.out file are used.

Alternative paths

These are the paths you specify with the *Set File Search Paths* option of the File menu.

For library files, if everything else failed, the search will be extended to the path(s) specified by the LIBPATH environment variable associated with the executable.

To specify alternative path(s), do the following:

- Select the *File* menu, and then the *Set File Search Paths* option. The *Alt File Search Path Dialog* window appears.
- Enter the name of the path in the *Enter Alt File Search Path(s)* text field. You can specify more than one path by separating each with colon (:) or a space.

  Notes:

  1. You can use the “at” symbol (@) with this option to represent the default file path, in order to specify that other paths be searched before the default path. For example, to set the alternative file search path(s) so that Xprofiler searches *pathA*, the default path, then *pathB*, type *pathA:@:pathB* in the *Alt File Search Path(s) (-a)* field.
  2. If @ is used in the alternative search path, the two buttons in the Alt File Search Path Dialog will be greyed out, and have no effect on the search order.
- Click on the OK button. The paths you specified in the text field become the alternative paths.

Changing the search sequence: You can change the order of the search sequence for library files and source code files via the *Set File Search Paths* option of the File menu. To change the search sequence, do the following:

1. Select the *File* menu, and then the *Set File Search Paths* option. The *Alt File Search Path Dialog* window appears.
2. To indicate the file search should use alternative paths first, click on the *Check alternative path(s)* first button.
3. Click on the OK button. This changes the search sequence to the:
   a. Alternative paths
   b. Default paths
   c. Path(s) specified in LIBPATH (library files only)

To return the search sequence back to its default order, repeat steps 1 through 3, but in step 2 above, click on the *Check default path(s) first* button. When the action is confirmed (by clicking on the OK button), the search sequence will start with the default paths again.

Keep in mind that if a file is found in one of the alternative paths or a path in LIBPATH, this path now becomes the default path for this file throughout the current Xprofiler session (until you exit this Xprofiler session or load a new set of data).

Understanding the Xprofiler display

The primary difference between Xprofiler and the UNIX® **gprof** command is that Xprofiler gives you a graphical picture of your application's CPU consumption in addition to textual data. This allows you to focus quickly on the areas of your application that consume a disproportionate amount of CPU.

Xprofiler displays your profiled program in a single main window. It uses several types of graphic images to represent the relevant parts of your program. Functions appear as solid green boxes (called *function boxes*), and the calls between them appear as blue arrows (called *call arcs*). The function boxes and call arcs that belong to each library within your application appear within a fenced-in area called a *cluster box*. The way that functions, calls, and library clusters are depicted is discussed later.

The Xprofiler main window

The Xprofiler main window contains a graphical representation of the functions and calls within your application as well as their inter-relationships. It provides six menus, including one for online help.

The Xprofiler main window looks similar to this when an application has been loaded:

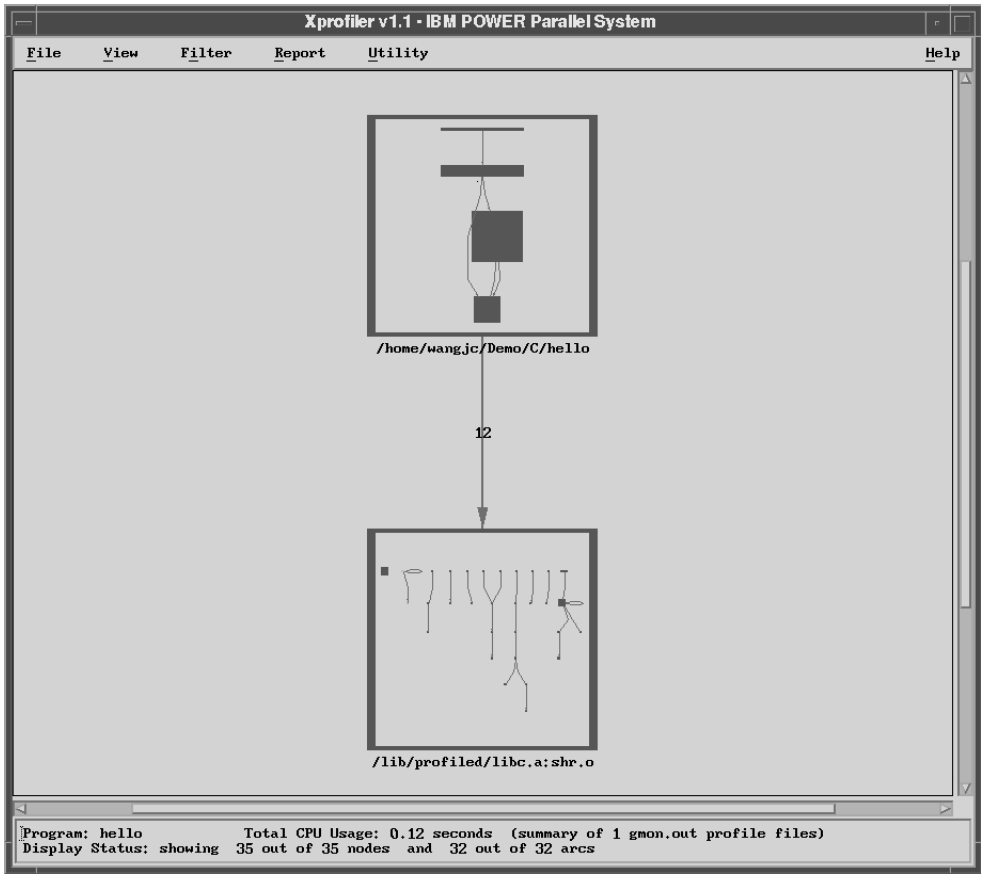


Figure 7. Sample Xprofiler Main Window

In the main window, Xprofiler displays the *function call tree*. The function call tree displays the function boxes, arcs, and cluster boxes that represent the functions within your application.

Note: When Xprofiler first opens, by default, the function boxes for your application will be *clustered* by library, as in the example above. This means that a cluster box appears around each library, and the function boxes and arcs within the cluster box are reduced in size. If you wish to see more detail, you need to uncluster the functions. To do this, select the *File* menu and then the *Uncluster Functions* option.

Xprofiler main menus

Along the upper portion of the main window is the *menu bar*. The left side of the menu bar contains the Xprofiler menus that let you work with your profiled data. On the right side of the menu bar, there is a Help menu for accessing online help.

The Xprofiler menus are described below:

File menu: The File menu lets you specify the executable (a.out) files and profile data (gmon.out) files that Xprofiler will use. It also lets you control how your files are accessed and saved.

View menu: The View menu lets you focus on specific portions of the function call tree in order to get a better view of the application's critical areas.

Filter menu: The Filter menu lets you add, remove, and change specific parts of the function call tree. By controlling what Xprofiler displays, you can focus on the objects that are most important to you.

Report menu: The Report menu provides several types of profiled data in a textual and tabular format. In addition to presenting the profiled data, the options of the Report menu let you:

- Display textual data
- Save it to a file
- View the corresponding source code
- Locate the corresponding function box or call arc in the function call tree.

Utility menu: The Utility menu contains one option; *Locate Function By Name*, which lets you highlight a particular function in the function call tree.

Xprofiler hidden menus

Function menu: The Function menu lets you perform a number of operations for any of the functions shown in the function call tree. You can access statistical data, look at source code, and control which functions get displayed.

The Function menu is not visible from the Xprofiler window. You access it by clicking on the function box of the function in which you are interested with your right mouse button. By doing this, you not only bring up the Function menu, but you select this function as well. Then, when you select actions from the Function menu, they are applied to this function.

Arc menu: The Arc menu lets you locate the caller and callee functions for a particular *call arc*. A call arc is the representation of a call between two functions within the function call tree.

The Arc menu is not visible from the Xprofiler window. You access it by clicking on the call arc in which you are interested with your right mouse button. By doing this, you not only bring up the Arc menu, but you select that call arc as well. Then, when you perform actions with the Arc menu, they are applied to that call arc.

Cluster Node menu: The Cluster Node menu lets you control the way your libraries are displayed by Xprofiler. In order to access the Cluster Node Menu, the function boxes, in the function call tree, must first be clustered by library. See “Clustering libraries together” on page 70 for information about clustering and unclustering the function boxes of your application. When the function call tree is clustered, all the function boxes within each library appear within a *cluster box*.

The Cluster Node menu is not visible from the Xprofiler window. You access it by clicking on the edge of the cluster box in which you are interested with your right mouse button. By doing this, you not only bring up the Cluster Node menu, but you select that cluster as well. Then, when you perform actions with the Cluster Node menu, they are applied to the functions within that library cluster.

Display Status field

At the bottom of the Xprofiler window is a single field that tells you:

- The name of your application
- The number of gmon.out files used in this session.
- The total amount of CPU used by the application.

- The number of functions and calls in your application, and how many of these are currently displayed

How functions are depicted

Functions are represented by green, solid-filled boxes in the function call tree. The size and shape of each function box indicates its CPU usage. The height of each function box represents the amount of CPU time it spent on executing itself. The width of each function box represents the amount of CPU time it spent executing itself, plus its descendant functions.

This type of representation is known as *summary mode*. In summary mode, the size and shape of each function box is determined by the total CPU time of multiple *gmon.out* files used on that function alone, and the total time used by the function and its descendant functions. A function box that is wide and flat represents a function that uses a relatively small amount of CPU on itself (it spends most of its time on its descendants). On the other hand, the function box for a function that spends most of its time executing only itself will be roughly square-shaped.

Functions can also be represented in *average mode*. In average mode, the size and shape of each function box is determined by the average CPU time used on that function alone, among all loaded *gmon.out* files, and the standard deviation of CPU time for that function among all loaded *gmon.out* files. The height of each function node represents the average CPU time, among all the input *gmon.out* files, used on the function itself. The width of each node represents the standard deviation of CPU time, among the *gmon.out* files, used on the function itself. The average mode representation is available only when more than one *gmon.out* file is entered. For more information on summary mode and average mode, see “Controlling the representation of the function call tree” on page 63.

Under each function box in the function call tree is a label that contains the name of the function and related CPU usage data. For information on the function box labels, see “Getting basic data” on page 75.

The example below shows the function boxes for two functions, *sub1* and *printf*, as they would appear in the Xprofiler display.

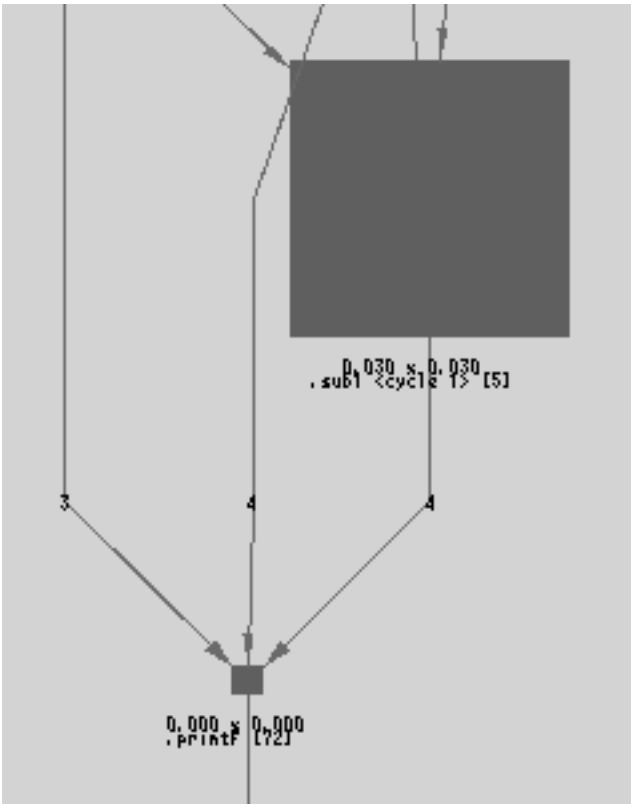


Figure 8. Example of Function Boxes and Arcs in Xprofiler Display

Each function box has its own menu. To access it, place your mouse cursor over the function box of the function in which you are interested, and press the right mouse button. Each function also has an information box that lets you get basic performance numbers quickly. To access the information box, place your mouse cursor over the function box of the function in which you are interested, and press the left mouse button.

How calls between functions are depicted

The calls made between each of the functions in the function call tree are represented by blue arrows extending between their corresponding function boxes. These lines are called *call arcs*. Each call arc appears as a solid blue line between two functions. The arrowhead indicates the direction of the call; the function represented by the function box 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*.

Each call arc includes a numerical label that tells you how many calls were exchanged between the two corresponding functions.

Figure 8, above, shows several call arcs. For the call arc that connects *sub1* and *printf*, *sub1* is the caller and *printf* is the callee. The label tells you that *sub1* called *printf* four times.

Note that each call arc has its own menu that lets you locate the function boxes for its caller and callee functions. To access it, place your mouse cursor over the call arc for the call in which you are interested, and press the right mouse button. Each call arc also has an information box that shows you the number of times the caller

function called the callee function. To access the information box, place your mouse cursor over the call arc for the call in which you are interested, and press the left mouse button.

How library clusters are depicted

Xprofiler lets you collect the function boxes and call arcs that belong to each of your shared libraries into *cluster boxes*. Figure 7 on page 49 shows an example of an Xprofiler display in which the libraries are clustered.

Since there will be a box around each library, the individual function boxes and call arcs will be difficult to see. If you want to see more detail, you will need to uncluster the function boxes. To do this, select the Filter menu and then the *Uncluster Functions* option.

When viewing function boxes within a cluster box, note that the size of each function box is relative to those of the other functions within the same library cluster. On the other hand, when all the libraries are unclustered, the size of each function box is relative to all the functions in the application (as shown in the function call tree).

Each library cluster has its own menu that lets you manipulate the cluster box. To access it, place your mouse cursor over the edge of the cluster box you are interested in, and press the right mouse button. Each cluster also has an information box that shows you the name of the library and the total CPU usage (in seconds) consumed by the functions within it. To access the information box, place your mouse cursor over the edge of the cluster box you are interested in and press the left mouse button.

Using the Xprofiler graphical user interface

The Xprofiler graphical user interface (GUI) contains features and buttons that are common throughout the interface. This section explains how to use some of these common elements.

Using the dialog window buttons

The buttons that appear on the Xprofiler dialog windows are explained below:

OK

Saves the changes, executes the action, and closes the dialog window.

Apply

Saves the changes, executes the action, but leaves the dialog window open.

Reset

Restores the fields of the dialog window to their original values (at the time you opened it), and keeps the dialog window open.

Cancel

Ignores changes and closes the dialog window.

Help

Brings up the Xprofiler online help.

Filter

Executes filtering criteria provided by you in the dialog window.

Using the search engine

Some of the Xprofiler windows that are accessible via the Report and Function menus provide a *Search Engine* field that lets you search for a specific string. In the Search Engine field, which is located at the bottom of these windows, you type the string in which you are interested. The first row that contains the string you specified is highlighted.

To use the Search Engine to search for a string:

1. Click on the *Search Engine* field with the left mouse button. The Search Engine field highlights to show that it is selected.
2. Type the string you are looking for in the Search Engine field.

   Extended regular expressions are allowed. For more information, see the explanation of the **regcmp** and **regcomp** commands in *AIX 5L Version 5.1 Technical Reference, Volume 2: Base Operating System and Extensions*.
3. Press the <**Enter**> key. The first row, in the Report or Source Code window, that contains the string you specified is highlighted. Each time you press the <**Enter**> key, a subsequent occurrence of the string in highlighted. The Search wraps back to the first occurrence after all other occurrences have been highlighted.

Using the save dialog windows

A *Save* dialog window appears when you choose the *Save As* option from any of the Xprofiler reports windows or from the File Menu. It allows you to save the data you see, in the report window that is currently open, to a file.

Note: If you choose the *Save As* option from one of the reports windows, the title of the dialog window included the name of the report (for example *Save Flat Profile*).

To save the current report data to a file using the Save dialog window:

1. Specify the file into which the data should be placed. You can specify either an existing file or a new one. If you specify and existing file, be aware that Xprofiler replaces the file altogether (instead of appending to the existing data). To replace an existing file, use the scroll bars of the *Directories* and the *Files* selection boxes to locate the file you want. To make locating your file easier, you can also use the *Filter* button (see “Using the dialog window filters” for more information). To specify a new file, type its name in the *Selection* field.
2. Click on the **OK** button. A file containing the profiled window data appears in the directory you specified, under the name you gave it.

Using the dialog window filters

Many of the Xprofiler dialog windows include a *Filter* button. The use of the Xprofiler Filter function follows the Motif standard. To use the Filter:

1. In the *Filter* field, specify the directory that contains the files that you wish to see in the *Files* selection box. You may specify an asterisk (*) as a wildcard.
2. Click on the *Filter* button with the left mouse button. The list of files in the Files selection box is updated to reflect your selection.

Using the Radio/Toggle buttons and sliders

Many of the dialog windows include buttons and sliders that allow you to select options and specify values.

Using the buttons

Aside from push-buttons, the Xprofiler dialog windows also use radio buttons and toggle buttons. Radio buttons let you select one item from a set of items, while toggle buttons let you activate or de-activate a single item.

In the example below, the *Screen Dump Options Dialog* window uses both radio buttons and toggle buttons. For instance, under *Output To*, there are two radio buttons; **File** and **Printer**. You must select one or the other, but you cannot select both. Just above the *Default Directory* field, notice two toggle buttons; **Enable Landscape** and **Annotate Output**. By using the toggle buttons, you can activate either one or both of these options.

To select (or activate) an option with a radio or toggle button, set the button to the pressed-in position by clicking on it. When a button is pressed-in, it appears shaded. Under *Output To*, in the example below, **File** is selected and **Printer** is not.

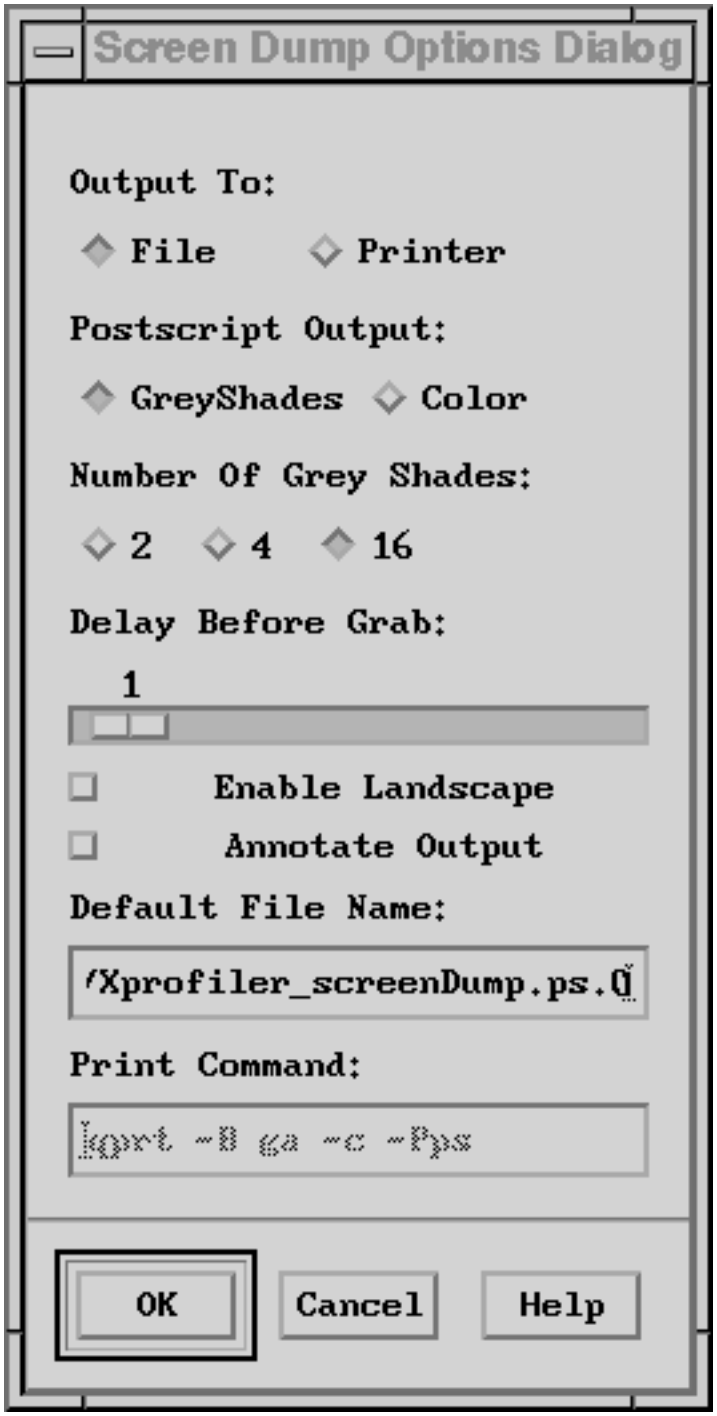


Figure 9. Example Showing Radio Buttons, Toggle Buttons, and Slider

Using the sliders

Several of the Xprofiler dialog windows include *sliders* that let you specify a numerical value. In the example above, the *Delay Before Grab* slider lets you specify the number of seconds you want to pass before the screen image is actually captured.

Place your mouse cursor over the slider. Press and hold the left mouse button while moving the slider horizontally in either direction. The number above the slider changes as you move it, and indicates the number selected. Once the slider is at the setting you want, release the mouse button.

If the number of selectable values on the slider is high, you may want to have finer control over the placement of the slider. If so, click on the slider and then use the arrow keys on your keyboard to place it.

Manipulating the function call tree

Xprofiler lets you look at your profiled data a number of ways, depending on what you want to see. It provides:

- Navigation that lets you move around the display and zoom in on specific areas
- Display options, based on your personal viewing preferences.
- Filtering capability, to let you include and exclude certain objects from the display

Zooming in on the function call tree

Xprofiler lets you magnify specific areas of the window to get a better view of your profiled data. The View menu includes three options that let you do this:

- Overview
- Zoom In
- Zoom Out

To resize a specific area of the display, you can use either the *Overview* or *Zoom In* options of the View menu. To magnify an area with the *Overview* option:

1. Select the *View* menu, and then the *Overview* option. The Overview Window appears, as in the example below.

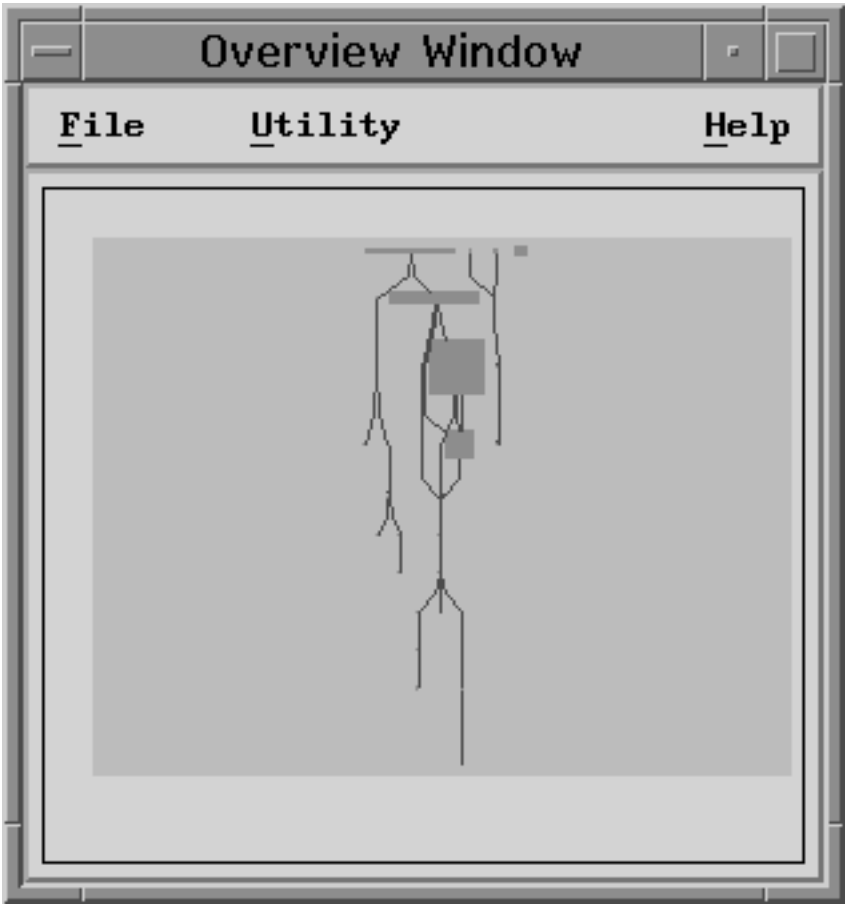


Figure 10. The Overview Window

The Overview Window contains a miniature view of the function call tree, just as it appears in the Xprofiler main display. When you open the Overview Window, the light blue highlight area represents the current view of the main window.

You control the size and placement of the highlight area with your mouse. Depending on where you place your mouse over the highlight area, your cursor changes to indicate the operation you can perform. Here is an explanation of the cursor images, and what they indicate to you:

When your cursor appears as two crossed arrows, it means that by pressing and holding your mouse button, you can control where the box is placed.

Figure 11. Cursor when movement of highlight box is under mouse control

When your cursor appears as a line with an arrow perpendicular to it, it means that your mouse button has grabbed the edge of the highlight area, and you now have the ability to resize it. By pressing and holding your mouse button, and dragging it

in or out, you can increase or decrease the size of the box. Notice that as you move the edge in or out, the size of the entire highlight area changes.

Figure 12. Cursor when edge of highlight box is under mouse control

When your cursor appears as a right angle with an arrow pointing into it, it means that your mouse button has grabbed the corner of the highlight area and you now have the ability to resize it. By pressing and holding your mouse button, and dragging it diagonally up or down, you can increase or decrease the size of the box. Notice that as you move the corner up or down, the size of the entire highlight area changes.

Figure 13. Cursor when corner of highlight box is under mouse control

3. Place your mouse cursor within the light blue highlight area. Notice that the cursor changes to four crossed arrows. This indicates that your cursor has control over the placement of the box.
4. Move your cursor over one of the four corners of the highlight area. Notice that the cursor changes to a right angle with an arrow pointing into it. This indicates that you now have control over the corner of the highlight area.
5. Press and hold the left mouse button, and drag the corner of the box diagonally inward. The box shrinks as you move it. The example below shows the highlight area reduced in size, with only a few function boxes visible within it.

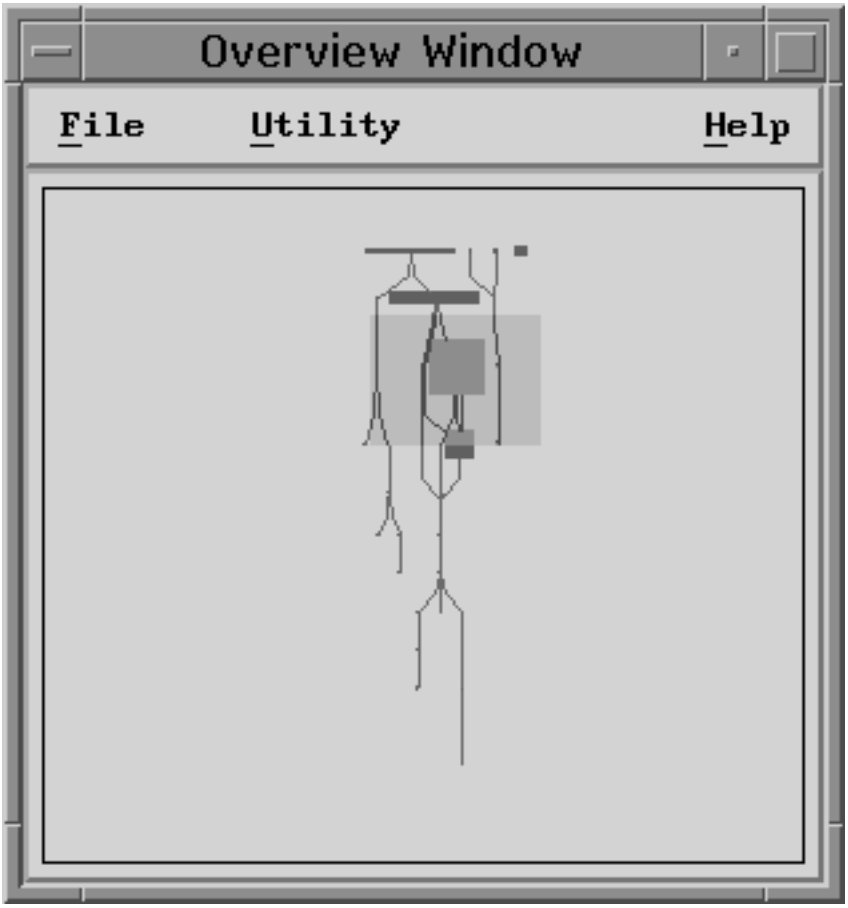


Figure 14. Highlight Area Reduced in Size

6. When the highlight area is as small as you would like it (or the smallest allowable size), release the mouse button. The Xprofiler main display redraws itself to contain only the functions within the highlight area, and in the same proportions. This has the effect of magnifying the items within the highlight area.
7. Place your mouse cursor over the highlight area. Your cursor again changes to four crossed arrows to indicate that you have control over the placement of the highlight area. Press and hold the left mouse button and drag the highlight area to the area of the Xprofiler display you want to magnify.
8. Release the mouse button. The Xprofiler main display now contains the items in which you are interested.

The example below shows the Xprofiler main display with the area, indicated by the highlight area in Figure 14, magnified. Note that the performance data, on the label of each function box, is now visible.

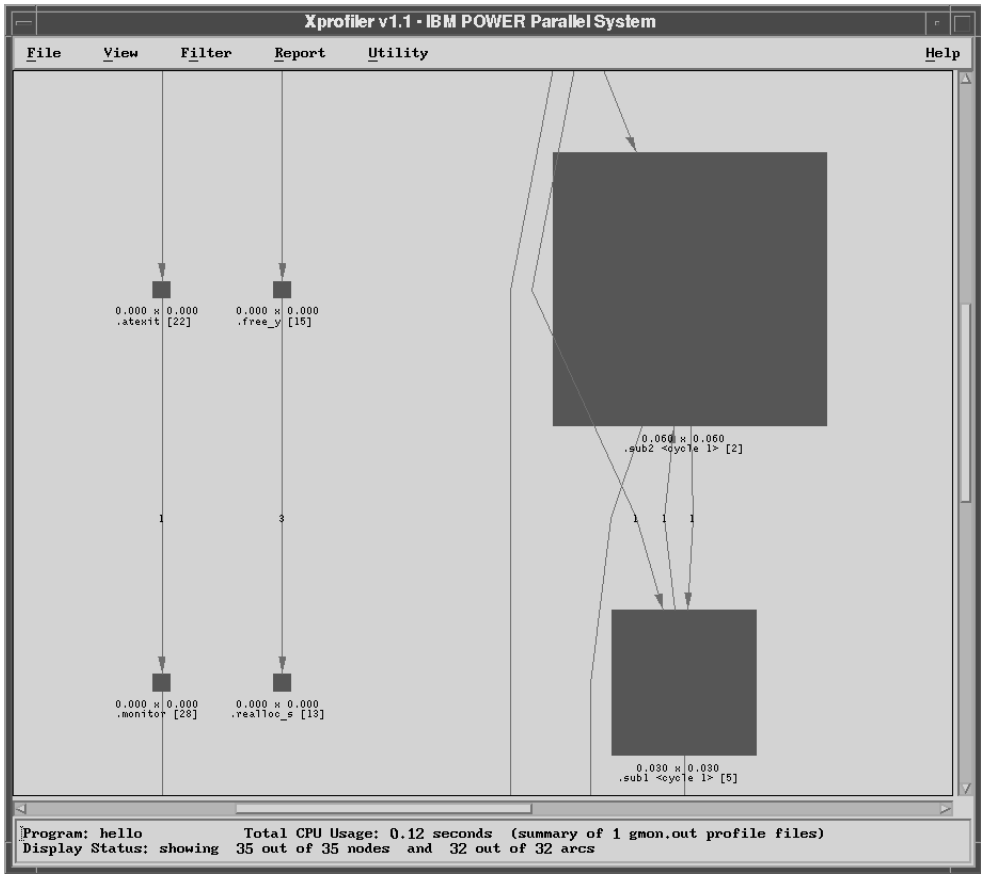


Figure 15. Magnified View of Xprofiler Display

To use the *Zoom In* option of the View menu to magnify a specific area of the function call tree:

1. Select the *View* menu, and then the *Zoom In* option. Once you select the *Zoom In* option, your cursor changes to a hand to indicate that your selection is active.
2. Place the mouse cursor in the upper left hand corner of the area you would like to view more closely. Press and hold the left mouse button while dragging it diagonally downward, until the rubber band box surrounds the area you want to view.
3. Release the mouse button. Xprofiler redraws the display so that the area of the function call tree you selected is centered and sized proportionately, according to the size of the rubber band box you drew.

To get an even closer view of the area you selected, choose the *Zoom In* option again and follow the steps above.

There may be times when you are looking at the function call tree too closely. The *Zoom Out* option lets you widen the view of the function call tree, as if you were taking a few steps back from a painting on a wall.

The *Zoom Out* option is most useful after using the *Zoom In* or *Overview* options to magnify an area of the function call tree. By default, the Xprofiler main window is completely zoomed out (it shows you the entire function call tree). The *Zoom Out* option helps you return the main window to this state.

To zoom out:

1. Select the *View* menu, and then the *Zoom Out* option. Once you select the *Zoom Out* option, your cursor changes to a hand to indicate that your selection is active.
2. Place the mouse cursor in the upper left hand corner of the area you want to view. Press and hold the left mouse button while dragging it diagonally downward, until the rubberband box surrounds the area you want to widen.
3. Release the mouse button. Xprofiler redraws the display so that the area of the function call tree you selected is centered and sized proportionately according to the size of the rubber band box that you drew.

To further step back from the area you selected, choose the *Zoom Out* option again, and follow the steps above.

Controlling how the display is updated

The Utility menu of the Overview window lets you choose the mode in which the display is updated. The default is the *Immediate Update* option, which causes the display to show you the items in the highlight area as you are moving it around. The *Delayed Update* option, on the other hand, causes the display to be updated only when you have moved the highlight area over the area in which you are interested, and released the mouse button. The *Immediate Update* option only applies to what you see when you move the highlight area; it has no effect on the resizing of items in highlight area, which is always delayed.

Other viewing options

Xprofiler lets you change the way it displays the function call tree, based on your own personal preferences.

Controlling the graphic style of the function call tree

You can choose between 2-D and 3-D function boxes in the function call tree. The default style is 2-D, but you can change this to 3-D. To do this:

1. Select the View menu, and then the *3-D Image* option. The function boxes in the function call tree now appear in 3-D format.

Controlling the orientation of the function call tree

You can choose to have Xprofiler display the function call tree in either *Top-to-Bottom* or *Left-to-Right* format. The default is Top-to-Bottom. If you would rather see the function call tree displayed in Left-to-Right format:

1. Select the View menu, and then the *Layout: Left→Right* option. The function call tree appears in Left-to-Right format.

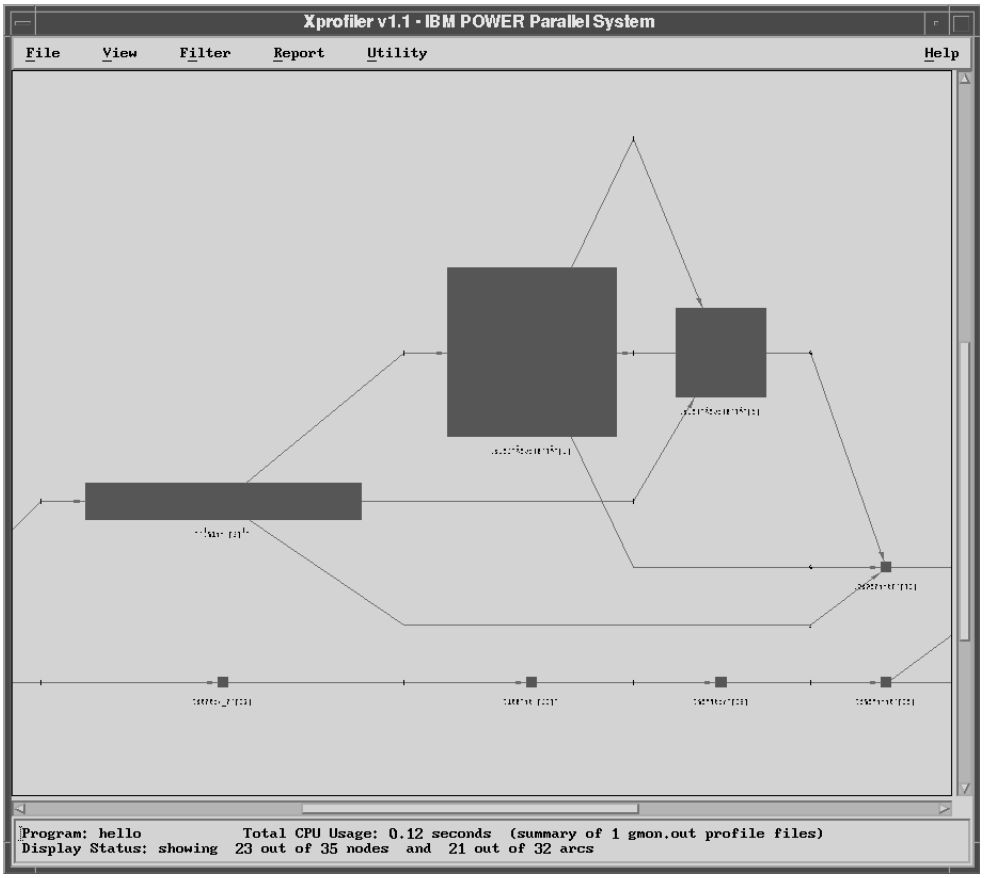


Figure 16. Left-to-Right Format

Controlling the representation of the function call tree

You can choose to have Xprofiler represent the function call tree in either *summary mode* or *average mode*.

When you select the *Summary Mode* option of the View menu, the size and shape of each function box is determined by the total CPU time of multiple *gmon.out* files used on that function alone, and the total time used by the function and its descendant functions. The height of each function node represents the total CPU time used on the function itself. The width of each node represents the total CPU time used on the function and its descendant functions. When the display is in summary mode, the *Summary Mode* option is greyed out and the *Average Mode* option is activated.

When you select the *Average Mode* option of the View menu, the size and shape of each function box is determined by the average CPU time used on that function alone, among all loaded *gmon.out* files, and the standard deviation of CPU time for that function among all loaded *gmon.out* files. The height of each function node represents the average CPU time, among all the input *gmon.out* files, used on the function itself. The width of each node represents the standard deviation of CPU time, among the *gmon.out* files, used on the function itself.

The purpose of average mode is to reveal workload balancing problems when an application is involved with multiple *gmon.out* files. In general, a function node with large standard deviation has a wide width, and a node with small standard deviation has a slim width.

Both summary mode and average mode only affect the appearance of the function call tree and the labels associated with it. All the performance data in Xprofiler reports and code displays are always summary data. If only one *gmon.out* file is given, both Summary Mode and Average Mode will be greyed out, and the display is always in Summary Mode.

Filtering what you see

When Xprofiler first opens, the entire function call tree appears in the main window. This includes the function boxes and call arcs that belong to your executable as well as the shared libraries that it utilizes. At times, you may want to simplify what you see in the main window, and there are a number of ways to do this.

Note: Filtering options of the Filter menu let you change the appearance of the function call tree only. The performance data contained in the reports (via the Reports menu) is not affected.

Restoring the status of the function call tree

Xprofiler allows you to undo operations that involve adding or removing nodes and arcs from the function call tree. When you undo an operation, you reverse the effect of any operation which adds or removes function boxes or call arcs to the function call tree. When you select the *Undo* option, the function call tree is returned to its appearance just prior to the performance of the add or remove operation. To undo an operation:

1. Select the *Filter* menu, and then the *Undo* option. The function call tree is returned to its appearance just prior to the performance of the add or remove operation.

Whenever you invoke the *Undo* option, the function call tree loses its zoom focus and zooms all the way out to reveal the entire function call tree in the main display. When you start Xprofiler, the *Undo* option is greyed out. It is activated only after an add or remove operation involving the function call tree takes place. After you undo an operation, the option greys out again until the next add or remove operation takes place.

The options that activate the *Undo* option include:

- In the main *File* menu:
- Load Configuration
- In the main *Filter* menu:
- Show Entire Call Tree
- Hide All Library Calls
- Add Library Calls
- Filter by Function Names
- Filter by CPU Time
- Filter by Call Counts
- In the *Function* menu:
- Immediate Parents
- All Paths To

- Immediate Children
- All Paths From
- All Functions on The Cycle
- Show This Function Only
- Hide This Function
- Hide Descendant Functions
- Hide This & Descendant Functions

If a dialog, like the Load Configuration Dialog or the Filter by CPU Time Dialog, is invoked and then cancelled immediately, the status of the *Undo* option is not affected. Once the option is available, it stays that way until you invoke it, or a new set of files is loaded into Xprofiler through the Load Files Dialog.

Displaying the entire function call tree

When you first open Xprofiler, by default, all the function boxes and call arcs of your executable and its shared libraries appear in the main window. After that, you may choose to filter out specific items from the window. However, there may be times when you want to see the entire function call tree again, without having to reload your application. To do this:

1. Select the *Filter* menu, and then the *Show Entire Call Tree* option. Xprofiler erases whatever is currently displayed in the main window and replaces it with the entire function call tree.

Excluding and including specific objects

There are a number of ways that Xprofiler lets you control the items that get displayed in the main window. For the most part, you will want to include or exclude certain objects so that you can more easily focus on the things that are of most interest to you.

Filtering shared library functions: In most cases, your application will call functions that are within shared libraries. By default, these shared libraries will appear in the Xprofiler window along with your executable. As a result, the window may get crowded and obscure the items that you really want to see. If this is the case, you may want to filter the shared libraries from the display. To do this:

1. Select the *Filter* menu, and then the *Remove All Library Calls* option.

The shared library function boxes disappear from the function call tree, leaving only the function boxes of your executable file visible.

If you removed the library calls from the display, you may want to add all or some of them back. To do this:

1. Select the *File* menu, and then the *Add Library Calls* option

The function boxes once again appear with the function call tree. Note, however, that all of the shared library calls that were in the initial function call tree may not be added back. This is because the *Add Library Calls* option only adds back in the function boxes for the library functions that were called by functions that are currently displayed in the Xprofiler window.

There may be times when you want to add only specific function boxes back into the display. To do this:

1. Select the *Filter* menu, and then the *Filter by Function Names* option. The *Filter By Function Names Dialog* window appears.

2. From the Filter By Function Names Dialog window, select the *add these functions to graph* button, and then type the name of the function you want to add in the *Enter function name* field. If you enter more than one function name, you must separate them by putting a blank space between each function name string.

   If there are multiple functions in your program that include the string you enter in their names, the filter applies to each one. For example, say you specified *sub*, and *print*, and your program also included functions named *sub1*, *psub1* and *printf*. The *sub*, *sub1*, *psub1*, *print*, and *printf* functions would all be added to the graph.
3. Click on the **OK** button. The function box(es) appears in the Xprofiler display with the function call tree.

Filtering by function characteristics: The Filter menu of Xprofiler offers you three options that allow you to add or subtract function boxes from the main window, based on specific characteristics. The options are:

- Filter by Function Names
- Filter by CPU Time
- Filter by Call Counts

Each one of these options uses a different dialog window to let you specify the criteria by which you want to include or exclude function boxes from the window.

To filter by function names:

1. Select the *Filter* menu.
2. Select the *Filter by Function Names* option. The *Filter By Function Names Dialog* window appears.

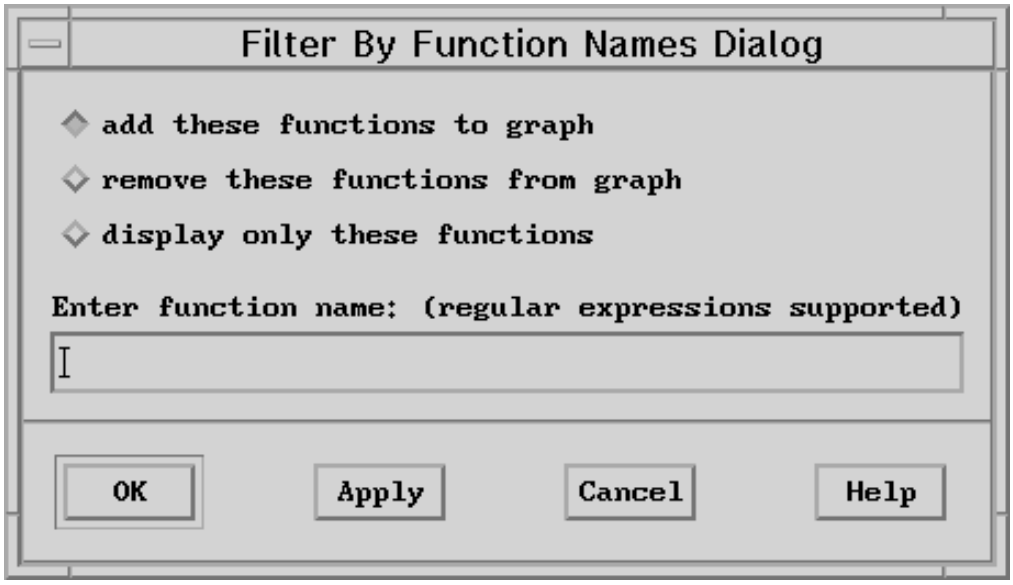


Figure 17. Filter By Function Names Dialog window

3. The Filter By Function Names Dialog window includes three options:
   – *add these functions to graph*
   – *remove these functions from the graph*
   – *display only these functions*

From the Filter By Function Names Dialog window, select the option you want, and then type the name of the function(s) to which you want it applied in the *Enter function name* field. For example, say you wanted to remove function box for a function called *printf*, from the main window. You would click on the *remove this function from the graph* button and type *printf* in the *Enter function name* field.

You can enter more than one function name in this field. If there are multiple functions in your program that include the string you enter in their names, the filter will apply to each one. For example, say you specified *sub* and *print*, and your program also included functions named *sub1*, *psub1*, and *printf*. The option you chose would be applied to the *sub*, *sub1*, *psub1*, *print*, and *printf* functions.
4. Click on the **OK** button. The contents of the function call tree now reflect the filtering options you specified.

To filter by CPU time:
1. Select the *Filter* menu.
2. Select the *Filter by CPU Time* option. The *Filter By CPU Time Dialog* window appears.

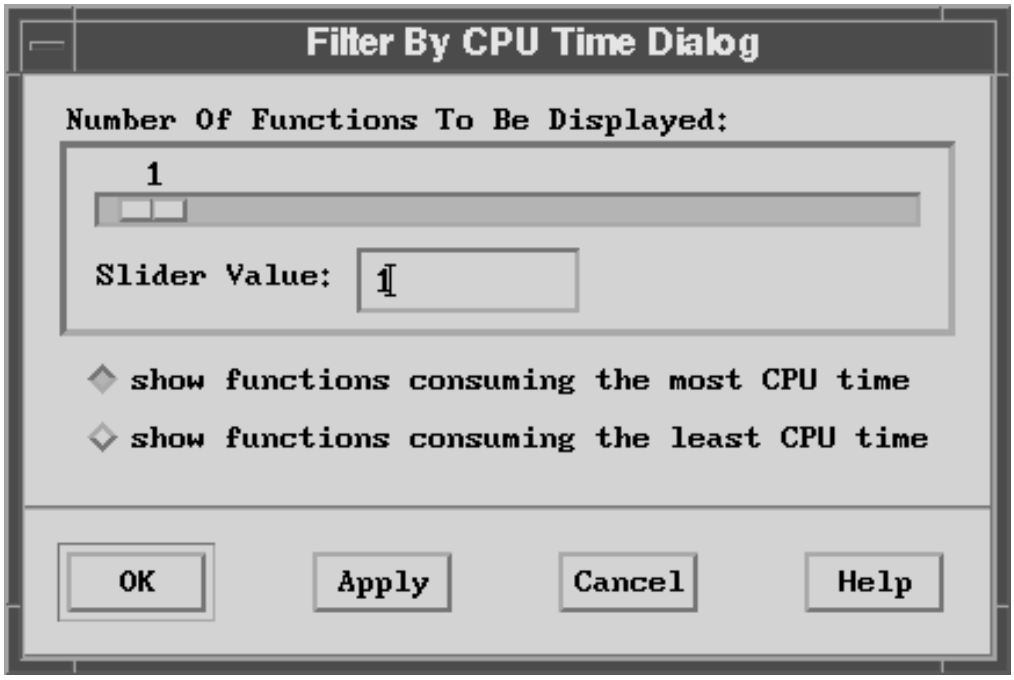


Figure 18. Filter By CPU Time Dialog window

3. The Filter By CPU Time Dialog window includes two options:
– *show functions consuming the most CPU time*
– *show functions consuming the least CPU time*
4. Click on the button for the option you want (*show functions consuming the most CPU time* is the default).
5. Select the number of functions to which you want it applied (1 is the default). You can move the slider in the *Functions* bar until the desired number appears, or you can enter the number in the *Slider Value* field. The slider and *Slider Value* field are synchronized so when the slider is updated, the text field value is updated also. If you enter a value in the text field, the slider is updated to that value when you click on the **Apply** button or the **OK** button.

For example, if you wanted to display the function boxes for the 10 functions in your application that consumed the most CPU, you would select the *show functions consuming the most CPU* button, and specify 10 with the slider or enter the value 10 in the text field.
6. Click on the **Apply** button to show the changes to the function call tree without closing the dialog. Click on the **OK** button to show the changes and close the dialog.

To filter by call counts:
1. Select the *Filter* menu.
2. Select the *Filter by Call Counts* option. The *Filter By Call Counts Dialog* window appears.

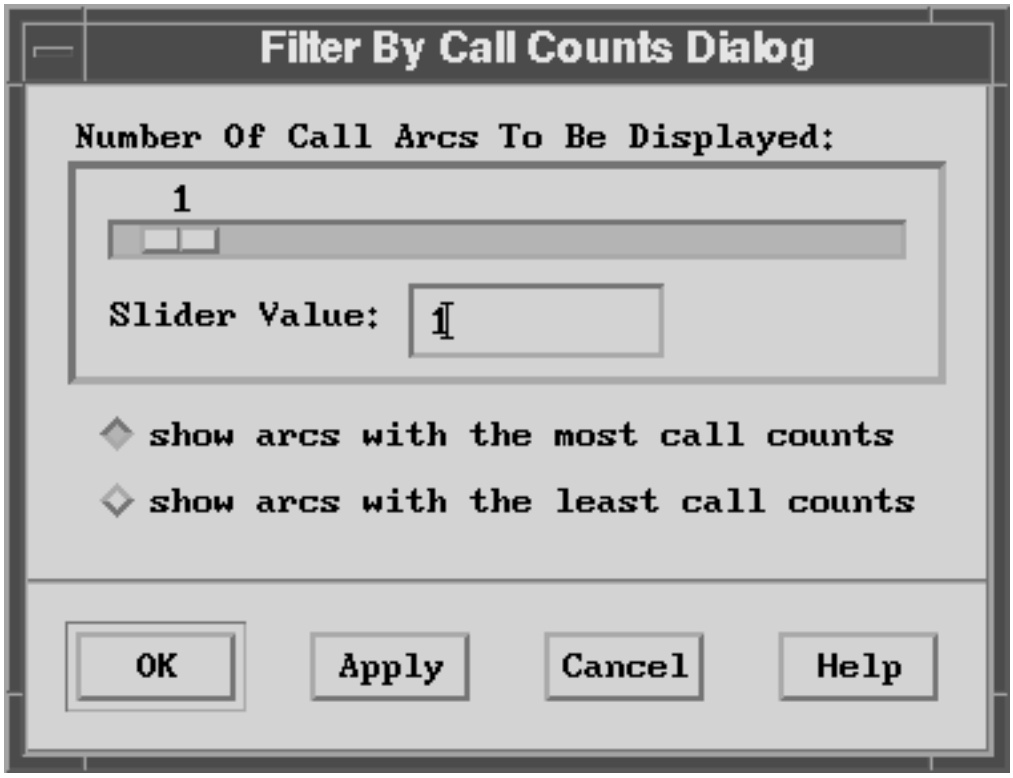


Figure 19. Filter By Call Counts Dialog window

3. The Filter By Call Counts Dialog window includes two options:
– *show arcs with the most call counts*
– *show arcs with the least call counts*
4. Click on the button for the option you want (*show arcs with the most call counts* is the default).
5. Select the number of call arcs to which you want it applied (1 is the default). You can move the slider in the *Call Arcs* bar until the desired number appears, or you can enter the number in the *Slider Value* field. The slider and *Slider Value* field are synchronized so when the slider is updated, the text field value is updated also. If you enter a value in the text field, the slider is updated to that value when you click on the **Apply** button or the **OK** button.

For example, if you wanted to display the 10 call arcs in your application that represented the least number of calls, you would select the *show arcs with the least call counts* button, and specify 10 with the slider or enter the value 10 in the text field.
6. Click on the **Apply** button to show the changes to the function call tree without closing the dialog. Click on the **OK** button to show the changes and close the dialog.

Including and excluding parent and child functions: When tuning the performance of your application, you will want to know which functions consumed the most CPU time, and then you will need to ask several questions in order to understand their behavior:

- Where did each function spend most of the CPU time?
- What other functions called this function? Were the calls made directly or indirectly?
- What other functions did this function call? Were the calls made directly or indirectly?

Once you understand how these functions behave, and are able to improve their performance, you can move on to analyzing the functions that consume less CPU.

When your application is large, the function call tree will also be large. As a result, the functions that are the most CPU-intensive may be difficult to see in the function call tree. To get around this, use the *Filter by CPU* option of the Filter menu, which lets you display only the function boxes for the functions that consume the most CPU time. Once you've done this, the Function menu for each function lets you add the parent and descendant function boxes to the function call tree. By doing this, you create a smaller, simpler function call tree that displays the function boxes associated with most CPU-intensive area of the application.

A *child* function is one that is directly called by the function of interest. To see only the function boxes for the function of interest and its child functions:

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The Function menu appears.
2. From the Function menu, select the *Immediate Children* option, and then the *Show Child Functions Only* option.

   Xprofiler erases the current display and replaces it with only the function boxes for the function you chose, plus its child functions.

A *parent* function is one that directly calls the function of interest. To see only the function box for the function of interest and its parent functions:

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The Function menu appears.
2. From the Function menu, select the *Immediate Parents* option, and then the *Show Parent Functions Only* option.

   Xprofiler erases the current display and replaces it with only the function boxes for the function you chose, plus its parent functions.

There may be times when you may want to see the function boxes for both the parent and child functions of the function in which you are interested, without erasing the rest of the function call tree. This is especially true if you chose to display the function boxes for two or more of the most CPU-intensive functions with the *Filter by CPU* option of the Filter menu (you suspect that more than one function is consuming too much CPU). To do this:

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The Function menu appears.
2. From the Function menu, select the *Immediate Parents* option, and then the *Add Parent Functions to Tree* option.

   Xprofiler leaves the current display as it is, but adds the parent function boxes.
3. Place your mouse cursor over the same function box and press the right mouse button. The Function menu appears.
4. From the Function menu, select the *Immediate Children* option, and then the *Add Child Functions to Tree* option.

   Xprofiler leaves the current display as it is, but now adds the child function boxes in addition to the parents.

Clustering libraries together

When you first bring up the Xprofiler window, by default, the function boxes of your executable, and the libraries associated with it, are clustered. Since Xprofiler shrinks the call tree of each library when it places it in a cluster, you will need to uncluster the function boxes if you want to look closely at a specific function box label.

It is important to understand that you can see significantly more detail per function, when your display is in the unclustered or *expanded* state, than when it is in the clustered or *collapsed* state. So, depending on what you want to do, you will need to cluster or uncluster (collapse or expand) the display.

There may be times when the Xprofiler window is visually crowded. This is especially true if your application calls functions that are within shared libraries; function boxes representing your executable functions as well as the functions of the shared libraries get displayed. As a result, you may want to organize what you see in the Xprofiler window so you can focus on the areas that are most important to you. One way you can do this is to collect all the function boxes of each library into a single area, known as a library *cluster*.

When you choose to cluster your libraries, Xprofiler gathers all the functions for each one into a single area, and draws a green box around them. This is known as a *cluster box*. The name of the library also appears below the box.

The function boxes in the application shown in Figure 7 on page 49 have been clustered. Note that one cluster box holds the function boxes associated with the executable *hello_world*, while the other cluster box holds the function boxes of the shared library */lib/profiled/libc.a:shr.o*.

The example below shows the same application, *hello_world*, with its function boxes unclustered. Now that the function boxes of your executable and its shared libraries are displayed together, which means their relationships are more apparent.

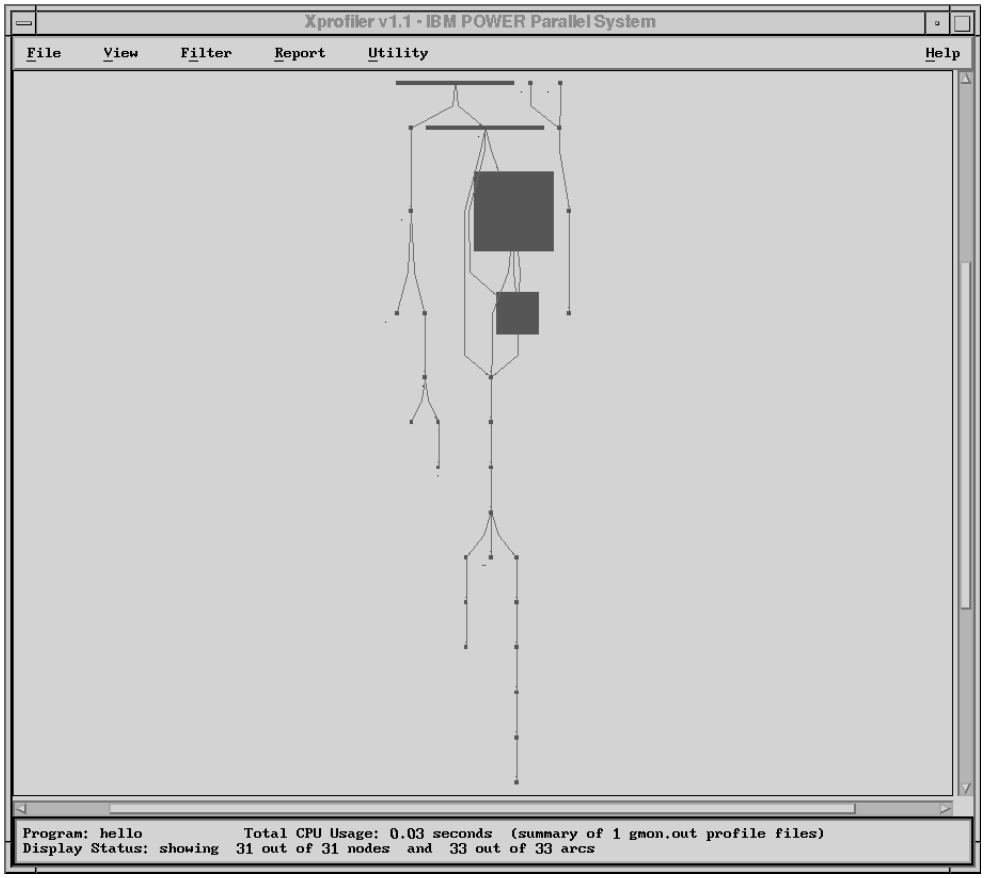


Figure 20. Xprofiler Window with Function Boxes Unclustered

Clustering functions

If the functions within your application are unclustered, you can use an option of the Filter menu to cluster them.

1. Select the Filter menu, and then the *Cluster Functions by Library* option. The libraries within your application appear within their respective cluster boxes.

Once you cluster the functions in your application, as shown in Figure 7 on page 49, you can further reduce the size (also referred to as *collapse*) of each cluster box. To do this:

1. Place your mouse cursor over the edge of the cluster box and press the right mouse button. The Cluster Node Menu appears.
2. Select the *Collapse Cluster Node* option. The cluster box, and its contents, now appear as a small solid green box. In the example below, the library */lib/profiled/libc.a:shr.o* is collapsed.

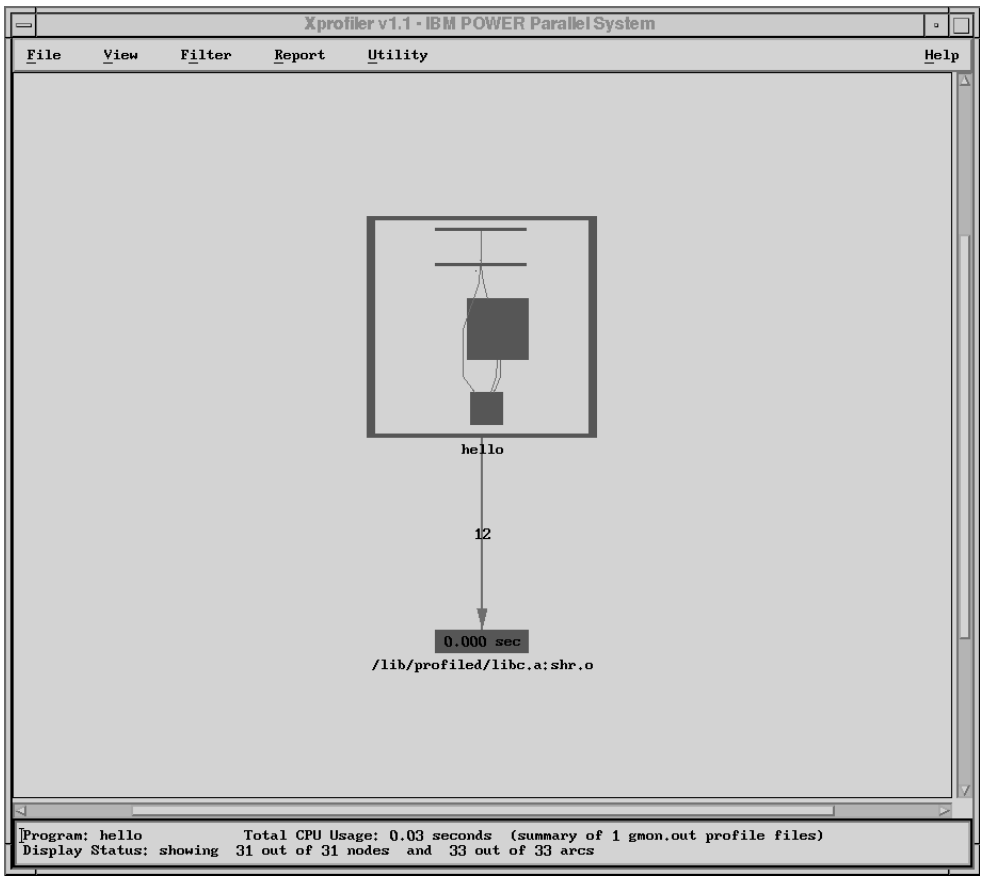


Figure 21. Xprofiler Window with One Library Cluster Box Collapsed

To return the cluster box to its original condition (*expand* it):

1. Place your mouse cursor over the collapsed cluster box and press the right mouse button. The Cluster Node Menu appears.
2. Select the *Expand Cluster Node* option. The cluster box, and its contents, appear once again.

Unclustering functions

If the functions within your application are clustered, you can use an option of the Filter menu to uncluster them.

1. Select the Filter menu, and then the *Uncluster Functions* option. The cluster boxes disappear and the functions boxes of each library expand to fill the Xprofiler window.

If your functions have been clustered, you may want to remove one or more (but not all) cluster boxes. For example, say you wanted to uncluster only the functions of your executable, but keep its shared libraries within their cluster boxes. You would:

1. Place your mouse cursor over the edge of the cluster box that contains the executable and press the right mouse button. The Cluster Node Menu appears.
2. Select the *Remove Cluster Box* option. The cluster box disappears and the function boxes and call arcs, that represent the executable functions, now appear in full detail. The function boxes and call arcs of the shared libraries remain within their cluster boxes, which now appear smaller to make room for the unclustered executable function boxes.

The example below shows the executable, *hello_world* with its cluster box removed. Its shared library, */lib/profiled/libc.a:shr.o*, remains within its cluster box.

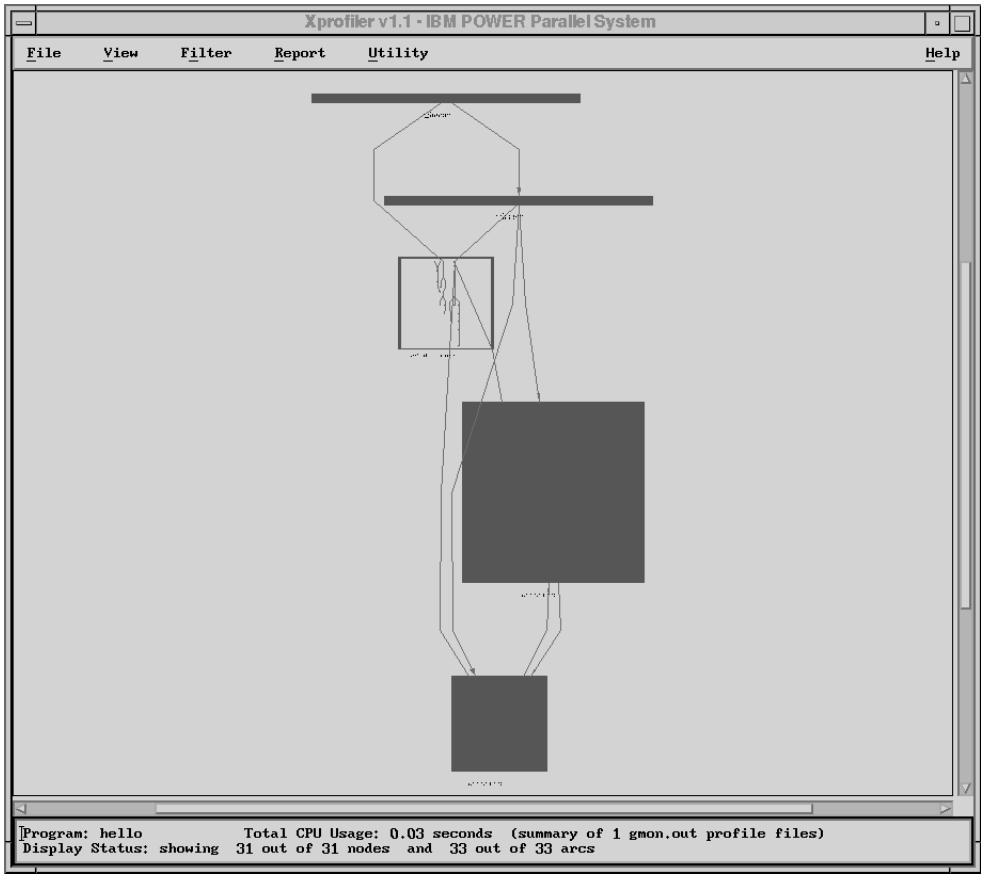


Figure 22. Xprofiler Window with One Library Cluster Box Removed

Locating specific objects in the function call tree

If you are interested in one or more specific functions in a complex program, you may need help locating their corresponding function boxes in the function call tree.

If you would like to locate a single function, and you know its name, you can use the *Locate Function By Name* option of the Utility menu. To locate a function by name:

1. Select the Utility menu, and then the *Locate Function By Name* option. The *Search By Function Name Dialog* window appears.
2. Type the name of the function you wish to locate in the *Enter Function Name* field. The function name you type here must be a continuous string (it cannot include blanks).
3. Press the **OK** or **Apply** button. The corresponding function box is highlighted (its color changes to red) in the function call tree and Xprofiler zooms in on its location.

   To display the function call tree in full detail again, go to the View menu and use the *Overview* option.

There may also be times when you want to see only the function boxes for the functions you are concerned with, plus other specific functions that are related to it. For instance, suppose you want to see all the functions that directly called the

function in which you are interested. It might not be easy to pick out these function boxes when you view the entire call tree, so you would want to display them, plus the function of interest, alone.

Each function has its own menu, called a *Function menu*. Via the Function menu, you can choose to see the following for the function in which you are interested:

- Parent functions (functions that directly call the function of interest)
- Child functions (functions that are directly called by the function of interest)
- Ancestor functions (functions that can call, directly or indirectly, the function of interest)
- Descendant functions (functions that can be called, directly or indirectly, by the function of interest)
- Functions that belong to the same cycle

When you use these options, Xprofiler erases the current display and replaces it with only the function boxes for the function of interest and all the functions of the type you specified.

Locating and displaying parent functions

A *parent* is any function that directly calls the function in which you are interested. To locate the parent function boxes of the function in which you are interested:

1. Click on the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select *Immediate Parents→Show Parent Functions Only*. Xprofiler redraws the display to show you only the function boxes for the function of interest and its parent functions.

Locating and displaying child functions

A *child* is any function that is directly called by the function in which you are interested. To locate the child functions boxes for the function in which you are interested:

1. Click on the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select *Immediate Children→Show Child Functions Only*. Xprofiler redraws the display to show you only the function boxes for the function of interest and its child functions.

Locating and displaying ancestor functions

An *ancestor* is any function that can call, directly or indirectly, the function in which you are interested. To locate the ancestor functions:

1. Click on the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select *All Paths To→Show Ancestor Functions Only*. Xprofiler redraws the display to show you only the function boxes for the function of interest and its ancestor functions.

Locating and displaying descendant functions

A *descendant* is any function that can be called, directly or indirectly, by the function in which you are interested. To locate the descendant functions (all the functions that the function of interest can reach, directly or indirectly):

1. Click on the function box of interest with the right mouse button. The Function menu appears.

2. From the Function menu, select *All Paths From→Show Descendant Functions Only*. Xprofiler redraws the display to show you only the function boxes for the function of interest and its descendant functions.

Locating and displaying functions on a cycle

To locate the functions that are on the same cycle as the function you are interested in:

1. Click on the function of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select *All Functions on the Cycle→Show Cycle Functions Only*. Xprofiler redraws the display to show you only the function of interest and all the other functions on its cycle.

Getting performance data for your application

With Xprofiler, you can get performance data for your application on a number of levels, and in a number of ways. You can easily view data pertaining to a single function, or you can use the reports provided to get information on your application as a whole.

Getting basic data

Xprofiler makes it easy to get data on specific items in the function call tree. Once you've located the item you are interested in, you can get data a number of ways. If you are having trouble locating a function in the function call tree, see "Locating specific objects in the function call tree" on page 73.

Basic function data

Below each function box in the function call tree is a label that contains basic performance data. The example below shows the function box for the function *sub1*, and its label.

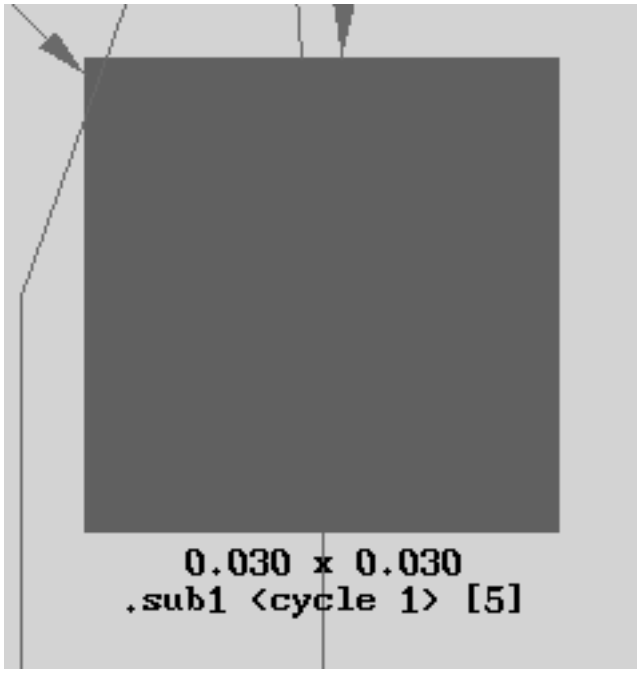


Figure 23. Example of a Function Box Label

The label contains the name of the function, its associated cycle, if any, and its index. In the example above, the name of the function is *sub1*. It is associated with cycle 1, and its index is 5. Also, depending on whether the function call tree is viewed in summary mode or average mode, the label will contain the information listed below. See “Controlling the representation of the function call tree” on page 63 for more about summary mode and average mode.

- In summary mode:
- The total amount of CPU time (in seconds) this function spent on itself plus the amount of CPU time it spent on its descendants (the number on the left of the *x*). In the example above, the function *sub1* spent .030 seconds on itself, plus its descendants.
- The amount of CPU time (in seconds) this function spent only on itself (the number on the right of the *x*). In the example above, the function *sub1* spent .030 seconds on itself.
- In average mode:
- The average CPU time (in seconds), among all the input *gmon.out* files, used on the function itself.
- The standard deviation of CPU time (in seconds), among all the input gmon.out files, used on the function itself.

Since labels are not always visible in the Xprofiler window when it is fully zoomed out, you may need to zoom in on it in order to see the labels. See “Zooming in on the function call tree” on page 57 for information on how to do this.

Basic call data

Call arc labels appear over each call arc. The label shows you the number of calls that were made between the two functions (from caller to callee). For example:

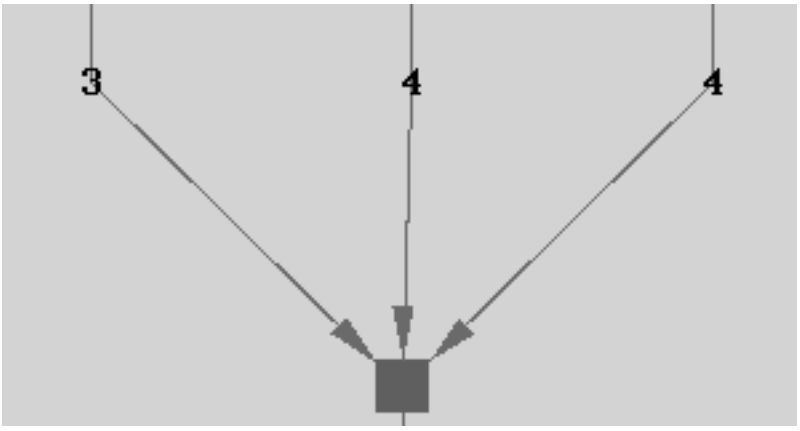


Figure 24. Example of a call arc label

In order to see a call arc label, you will probably need to zoom in on it. See “Zooming in on the function call tree” on page 57 for information on how to do this.

Basic cluster data

Cluster box labels tell you the name of the library that is represented by that cluster. If it is a shared library, the label shows its full pathname.

Information boxes

For each function box, call arc, and cluster box, there is a corresponding information box that you can access with your mouse. It gives you the same basic data that appears on the label. This is useful when the Xprofiler display is fully

zoomed out and the labels are not visible. To access the information box, click on the function box, call arc, or cluster box (place it over the edge of the box) with the left mouse button. The information box appears.

For a function, the information box contains:

- The name of the function, its associated cycle, if any, and its index.
- The amount of CPU used by this function. There are two values supplied in this field. The first is the amount of CPU time spent on this function plus the time spent on its descendants. The second value represents the amount of CPU time this function spent only on itself.
- The number of times this function was called (by itself or any other function in the application).

  For a call, the information box contains:

  – The caller and callee functions (their names) and their corresponding indexes.
  – The number of times the caller function called the callee.

  For cluster, the information box contains:

- The name of the library
- The total CPU usage (in seconds) consumed by the functions within it.

Function menu Statistics Report option

You can get performance statistics for a single function via the *Statistics Report* option of the Function menu. It lets you see data on the CPU usage and call counts of the selected function. If you are using more than one gmon.out file, this option breaks down the statistics per each gmon.out file you use.

When you select the *Statistics Report* menu option, the *Function Level Statistics Report* window appears, as in the example below.

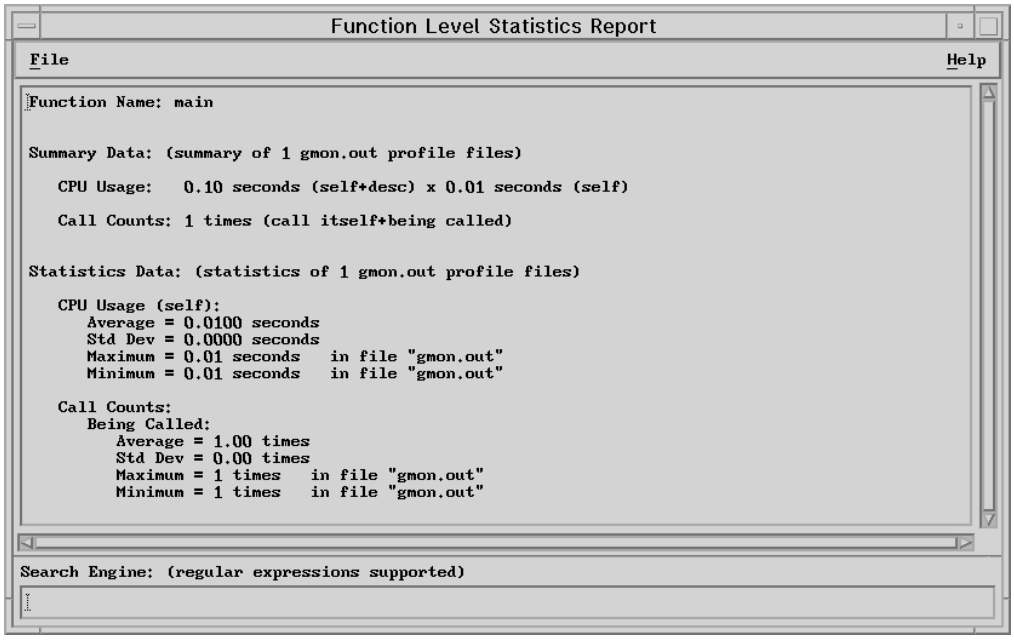


Figure 25. Function Level Statistics Report window

The Function Level Statistics Report window provides the following information:

Function Name
The name of the function you selected. In Figure 25 on page 77, the function name is *main*.

Summary Data
The total amount of CPU used by this function. If you used multiple gmon.out files, the value shown here represents their sum.

The *CPU Usage* field shows you:

- The amount of CPU used by this function. There are two values supplied in this field. The first is the amount of CPU time spent on this function plus the time spent on its descendants. The second value represents the amount of CPU time this function spent only on itself.

  In Figure 25 on page 77, CPU usage is listed as *0.10 seconds (self+desc) x 0.10 seconds (self)*

The *Call Counts* field shows you:

- The number of times this function called itself, plus the number of times it was called by other functions.

  In Figure 25 on page 77, the value in the Call Counts field is *1 times*.

Statistics Data
The CPU usage and calls made to or by this function, broken down by gmon.out file.

The *CPU Usage* field shows you:

- Average

  The average CPU time used by the data in each gmon.out file. In Figure 25 on page 77, the Average is listed as *0.0100 seconds*.
- Std Dev

  Standard deviation. A value that represents the difference in CPU usage samplings, per function, from one gmon.out file to another. The smaller the standard deviation, the more balanced the workload. In Figure 25 on page 77, the Std Dev is listed as *0.0000 seconds*.
- Maximum

  Of all the gmon.out files, the maximum amount of CPU time used. The corresponding gmon.out file appears to the right. In Figure 25 on page 77, the Maximum is listed as *0.01 seconds*.
- Minimum

  Of all the gmon.out files, the minimum amount of CPU time used. The corresponding gmon.out file appears to the right. In Figure 25 on page 77, the Minimum is listed as *0.01 seconds*.

The *Call Counts* field shows you:

- Average

  The average number of calls made to this function or by this function, per gmon.out file. In Figure 25 on page 77, the Average is *1.00 times*.
- Std Dev

  Standard deviation. A value that represents the difference in call count sampling, per function, from one gmon.out file to another. A small standard deviation value

in this field means that the function was almost always called the same number of times in each gmon.out file. In Figure 25 on page 77, the Std Dev is *0.00 times*.

- Maximum

  The maximum number of calls made to this function or by this function in a single gmon.out file. The corresponding gmon.out file appears to the right. In Figure 25 on page 77, the Maximum is *1 times*.

- Minimum

  The minimum number of calls made to this function or by this function in a single gmon.out file. The corresponding gmon.out file appears to the right. In Figure 25 on page 77, the Minimum is *1 times*.

Getting detailed data via reports

Xprofiler provides performance data in textual and tabular format. This data is provided in various tables called *reports*. If you are a **gprof** user, you are familiar with the *Flat Profile*, *Call Graph Profile*, and *Function Index* reports. Xprofiler generates these same reports, in the same format, plus two others.

You can access the Xprofiler reports from the Report menu. The Report menu lets you see the following reports:

- Flat Profile
- Call Graph Profile
- Function Index
- Function Call Summary
- Library Statistics

Each report window includes a File menu. Under the File menu is the *Save As* option which allows you to save the report to a file. For information on using the *Save File Dialog* window to save a report to a file, see “Using the save dialog windows” on page 54.

Each report window also includes a *Search Engine* field, which is located at the bottom of the window. The Search Engine lets you search for a specific string in the report. For information on using the Search Engine field, see “Using the search engine” on page 54.

Note: If you select the *Save As* option from the Flat Profile, Function Index, or Function Call Summary report windows, you must either complete the save operation or cancel it before you can select any other option from the menus of these reports. You can, however, use the other menus of Xprofiler before completing the save operation or canceling it, with the exception of the Load Files option, of the File menu, which remains greyed out.

Each of the Xprofiler reports are explained below.

Flat Profile report

When you select the *Flat Profile* menu option, the *Flat Profile* window appears. The Flat Profile report shows you the total execution times and call counts for each function (including shared library calls) within your application. The entries for the functions that use the greatest percentage of the total CPU usage appear at the top of the list, while the remaining functions appear in descending order, based on the amount of time used.

Unless you specified the **-z** command line option, the Flat Profile report does not include functions whose CPU usage is 0 (zero) and have no call counts.

Note that the data presented in the Flat Profile window is the same data that is generated with the UNIX **gprof** command.

The Flat Profile report looks similar to this:

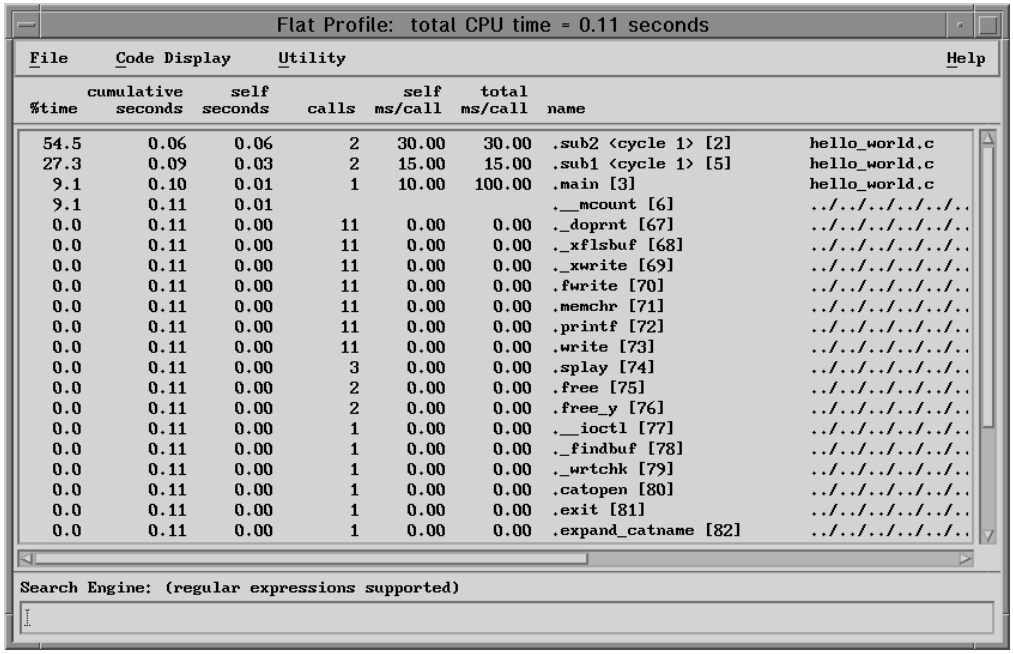


Figure 26. Flat Profile Report

Flat Profile window fields: The Flat Profile window fields are explained below.

- %time

  The percentage of the program's total CPU usage that is consumed by this function.

- cumulative seconds

  A running sum of the number of seconds used by this function and those listed above it.

- self seconds

  The number of seconds used by this function alone. The *self seconds* values are what Xprofiler uses to sort the functions of the Flat Profile report.

- calls

  The number of times this function was called (if this function is profiled). Otherwise, it is blank.

- self ms/call

  The average number of milliseconds spent in this function per call (if this function is profiled). Otherwise, it is blank.

- total ms/call

  The average number of milliseconds spent in this function and its descendants per call (if this function is profiled). Otherwise, it is blank.

- name

The name of the function. The *index* appears in brackets [] to the right of the function name. The index serves as the function's identifier within Xprofiler. It also appears below the corresponding function in the function call tree.

Call Graph Profile report

The *Call Graph Profile* menu option lets you view the functions of your application, sorted by the percentage of total CPU usage that each function, and its descendants, consumed. When you select this option, the *Call Graph Profile* window appears.

Unless you specified the **-z** command line option, the Call Graph Profile report does not include functions whose CPU usage is 0 (zero) and have no call counts.

Note that the data presented in the Call Graph Profile window is the same data that is generated with the UNIX **gprof** command.

The Call Graph Profile report looks similar to this:

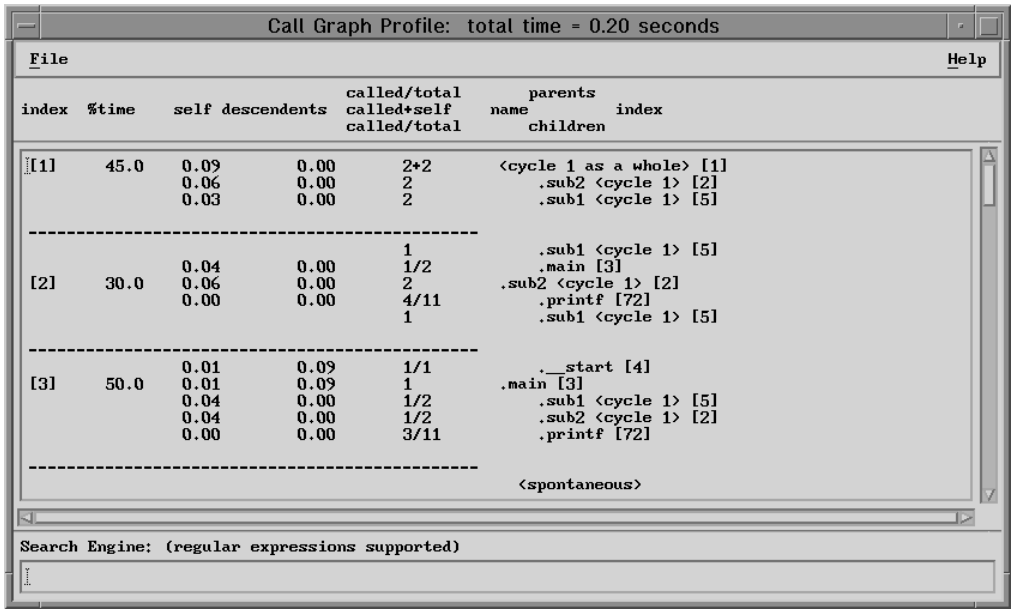


Figure 27. Call Graph Profile Report

Call Graph Profile window fields: The fields of the Call Graph Profile are explained below.

- index

  The index of the function in the Call Graph Profile. Each function in the Call Graph Profile has an associated index number which serves as the function's identifier. The same index also appears with each function box label in the function call tree, as well as other Xprofiler reports.

- %time

  The percentage of the program's total CPU usage that was consumed by this function and its descendants.

- self

  The number of seconds this function spends within itself.

- descendants

The number of seconds spent in the descendants of this function, on behalf of this function.

- called/total, called+self, called/total

  The heading of this column refers to the three different kinds of calls that take place within your program. The values in this field correspond to the functions listed in the *name, index, parents, children* field to its right. Depending on whether the function is a parent, child, or the function of interest (the function who's index is listed in the *index* field of this row), this value can stand for one of the following:

  – Number of times a parent called the function of interest
  – Number of times the function of interest called itself, recursively
  – Number of times the function of interest called a child

In the example below, *sub2* is the function of interest, *sub1* and *main* are its parents, and *printf* and *sub1* are its children.

```
called/total       parents
called+self    name           index
called/total       children

---------------
      1              .sub1 <cycle 1> [5]
      1/2            .main [3]
      2           .sub2 <cycle 1> [2]
      4/11           .printf [72]
      1              .sub1 <cycle 1> [5]
```

Figure 28. called/total, call/self, called/total field

- called/total

  For a parent function, this refers to the number of calls made to the function of interest, as well as the total number of calls it made to all functions. In the example above, one of the parent functions, *main*, made two calls; one to the function of interest, *sub2*, and one to another function.

- called+self

  This refers to the number of times the function of interest called itself, recursively. In the example above, the function of interest, *sub2*, called itself two times. For a child function, this refers to the number of times the function of interest called this child. In the example above, one of the child functions, *printf*, was called eleven times; four times by the function of interest, *sub2*, and seven times by other functions.

- name, index, parents, children

  The layout of the heading of this column is indicative of the information that is provided. To the left is the name of the function, and to its right is the function's index number. Appearing above the function are its parents, and below are its children.

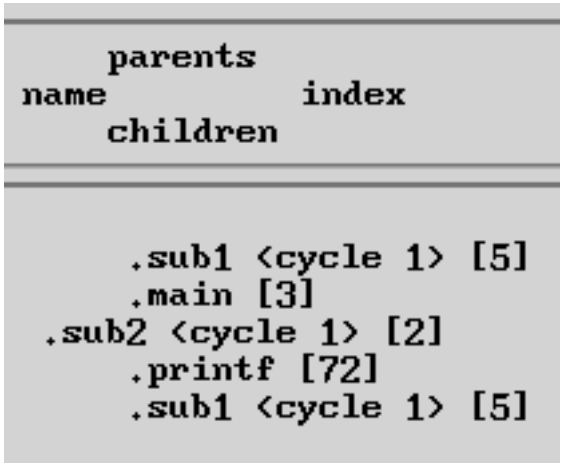


Figure 29. name/index/parents/children field

– name

  The name of the function, with an indication of its membership in a cycle, if any. Note that the function of interest appears to the left, while its parent and child functions are indented above and below it. In the example above, the name of the function is *sub2*.

– index

  The index of the function in the Call Graph Profile. This number corresponds to the index that appears in the *index* column of the Call Graph Profile and the on the function box labels in the function call tree. In the example above, the index of *sub2* is [2].

– parents

  The parents of the function. A *parent* is any function that directly calls the function in which you are interested. In the example above, the parents are *sub1* and *main*.

  If any portion of your application was not compiled with the **-pg** option, Xprofiler will not be able to identify the parents for the functions within those portions. As a result, these parents will be listed as *spontaneous* in the Call Graph Profile report.

– children

  The children of the function. A *child* is any function that is directly called by the function in which you are interested. In the example above, the children are *printf* and *sub1*.

Function Index report

The *Function Index* menu option lets you view a list of the function names included in the function call tree. When you select this option, the *Function Index* window appears, and displays the function names in alphabetical order. To the left of each function name is its *index*, enclosed in brackets []. The index is the function's identifier, which is assigned by Xprofiler. An index also appears on the label of each corresponding function box in the function call tree as well as other reports.

Unless you specified the **-z** command line option, the Function Index report does not include functions whose CPU usage is 0 (zero) and have no call counts.

The *Function Index* menu option includes a *Code Display* menu, like the *Flat Profile* menu option, allowing you to view source code or disassembler code. For more information on viewing code, see “Viewing source code” on page 89 and “Viewing disassembler code” on page 90.

The Function Index report looks similar to this:

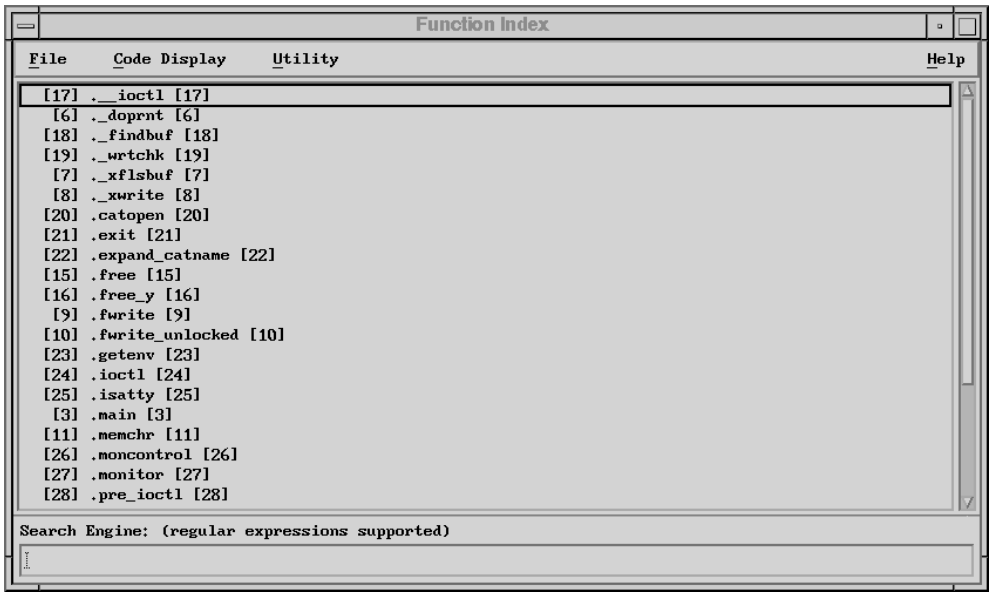


Figure 30. Sample Function Index Report

Function Call Summary report

The *Function Call Summary* menu option lets you display all the functions in your application that call other functions. They appear as caller-callee pairs (call arcs, in the function call tree), and are sorted by the number of calls in descending order. When you select this option, the *Function Call Summary* window appears.

The Function Call Summary report looks similar to this:

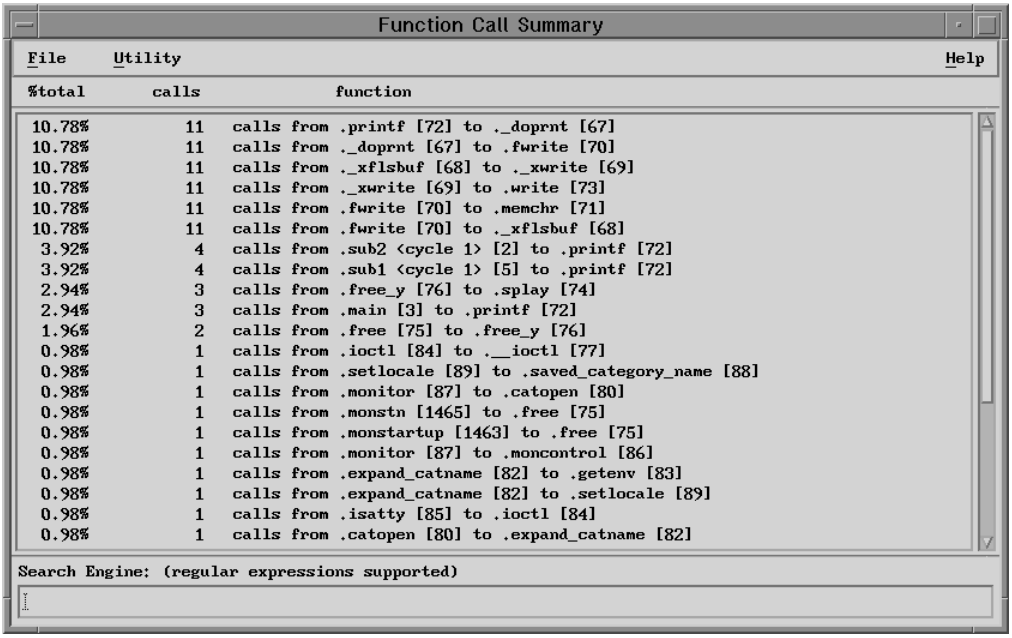


Figure 31. Sample Function Call Summary Report

Function Call Summary window fields: The fields of the Function Call Summary window are explained below.

- %total

  The percentage of the total number of calls generated by this caller-callee pair.
- calls

  The number of calls attributed to this caller-callee pair.
- function

  The name of the caller function and callee function.

Library Statistics report

The *Library Statistics* menu option lets you display the CPU time consumed and call counts of each library within your application. When you select this option, the *Library Statistics* window appears.

The Library Statistics report looks similar to this:

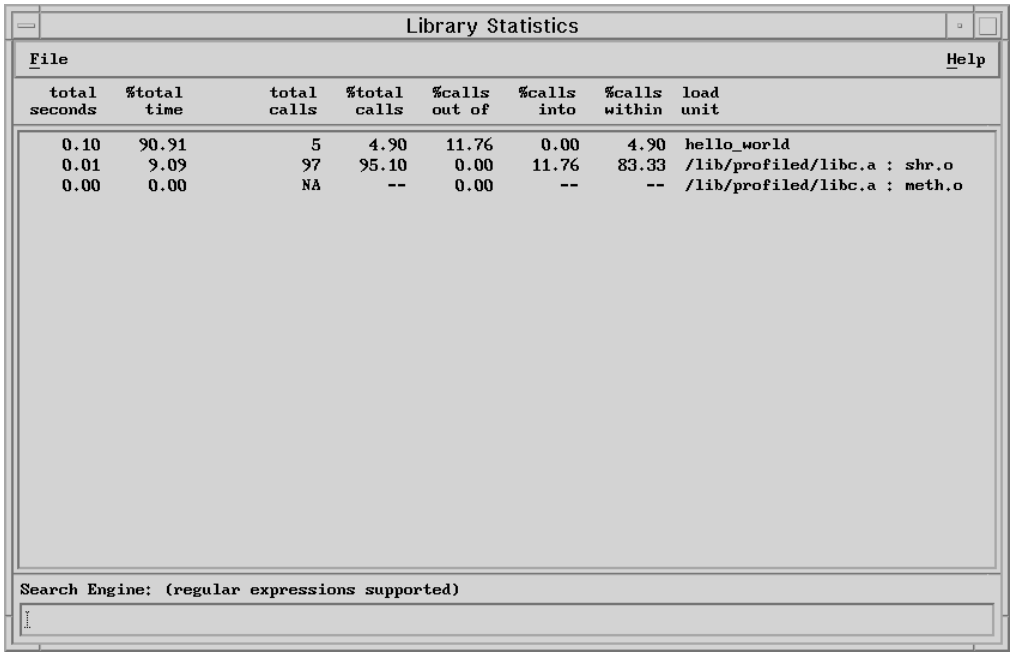


Figure 32. Sample Library Statistics Report

Library Statistics window fields: The fields of the Library Statistics window are explained below.

- total seconds

  The total CPU usage of the library, in seconds.
- %total time

  The percentage of the total CPU usage that was consumed by this library.
- total calls

  Total number of calls generated by this library.
- %total calls

  The percentage of the total calls that were generated by this library.
- %calls out of

  The percentage of the total number of calls made from this library to other libraries.
- %calls into

  The percentage of the total number of calls made from other libraries into this library.
- %calls within

  The percentage of the total number of calls made between the functions within this library.
- load unit

  The library's full path name.

Saving reports to a file

Xprofiler lets you save any of the reports you generate with the Report menu to a file. You can do this via the File and Report menus of the Xprofiler GUI.

Saving a single report: To save a single report, go to the Report menu, on the Xprofiler main window, and select the report you would like to save. Each report window includes a File menu. Select the File menu and then the *Save As* option to

save the report. A *Save* dialog window appears, which is named according to the report from which you selected the *Save As* option. For instance, if you chose *Save As* from the Flat Profile window, the dialog window is named *Save Flat Profile Dialog*.

Saving the Call Graph Profile, Function Index, and Flat Profile reports to a file: You can save the Call Graph Profile, Function Index, and Flat Profile reports to a single file with the File menu of the Xprofiler main window. The information you generate here is identical to the output of the UNIX **gprof** command. From the File menu, select the *Save As* option. The *Save File Dialog* window appears.

To save the report(s):

1. Specify the file into which the profiled data should be placed. You can specify either an existing file or a new one. To specify an existing file, use the scroll bars of the *Directories* and the *Files* selection boxes to locate the file you want. To make locating your files easier, you can also use the *Filter* button (see “Using the dialog window filters” on page 54 for more information). To specify a new file, type its name in the *Selection* field.
2. Click on the **OK** button. A file containing the profiled data appears in the directory you specified, under the name you gave it.

Note: Once you select the *Save As* option from the File menu, and the *Save Profile Reports* window opens, you must either complete the save operation or cancel it before you can select any other option from the menus of its parent window. For example, if you select the *Save As* option from the Flat Profile report, and the *Save File Dialog* window appears, you cannot use any other option of the Flat Profile report window.

The *File Selection* field of the *Save File Dialog* window follows Motif standards.

Saving summarized data from multiple profile data files: If you are profiling a parallel program, you could specify more than one profile data (gmon.out) file when you started Xprofiler. The *Save gmon.sum As* option of the File menu lets you save a summary of the data in each of these files to a single file.

The Xprofiler *Save gmon.sum As* option produces the same result as the Xprofiler and **gprof -s** command line option. If you run Xprofiler later, you can use the file you create here as input with the **-s** option. In this way, you can accumulate summary data over several runs of your application.

To create a summary file:

1. Select the File menu, and then the *Save gmon.sum As* option. The *Save gmon.sum Dialog* window appears.
2. Specify the file into which the summarized, profiled data should be placed. By default, Xprofiler puts the data into a file called *gmon.sum*, but you can designate a different file. You can either specify a new file or an existing one. To specify a new file, type its name in the selection field. To specify an existing file, use the scroll bars of the *Directories* and *Files* selection boxes to locate the file you want. To make locating your files easier, you can also use the Filter button (see “Using the dialog window filters” on page 54 for information).
3. Click on the OK button. A file, containing the summary data, appears in the directory you specified, under the name you gave it.

Saving a configuration file: The *Save Configuration* menu option lets you save the names of the functions that are displayed currently to a file. Later, in the same

Xprofiler session or a different session, you can read in this configuration file using the *Load Configuration* option. See the following section, “Loading a Configuration File”, for more information.

To save a configuration file:

1. Select the File menu, and then the *Save Configuration* option. The *Save Configuration File Dialog* window opens with the *program.cfg* file as the default value in the *Selection* field. “Program” is the name of the input *a.out* file.

   You can use the default file name, enter a file name in the *Selection* field, or select a file from the dialog’s files list.
2. Specify a file name in the *Selection* field and click on the **OK** button. A configuration file is created containing the name of the program and the names of the functions that are displayed currently.
3. Specify an existing file name in the *Selection* field and click on the **OK** button. An *Overwrite File Dialog* window appears so you can check the file before overwriting it.

If you select the *Forced File Overwriting* option in the *Runtime Options Dialog* window, the *Overwrite File Dialog* does not open and the specified file is overwritten without warning.

Loading a configuration file: The *Load Configuration* menu option lets you read in a configuration file that you saved. See the previous section, “Saving a Configuration File”, for more information. The *Load Configuration* option automatically reconstructs the function call tree according to the function names recorded in the configuration file.

To load a configuration file:

1. Select the File menu, and then the *Load Configuration* option. The *Load Configuration File Dialog* window opens. If a configuration files were loaded previously during the current Xprofiler session, the name of the file that was most recently loaded will appear in the **Selection** field of this dialog.

   You can also load the file with the **-c** command line option. See “Specifying command line options (from the GUI)” on page 43 for more information.
2. Select a configuration file from the dialog’s **Files** list or specify a file name in the **Selection** field, and click on the **OK** button. The function call tree is redrawn to show only those function boxes for functions that are listed in the configuration file and are called within the program that is currently represented in the display. All corresponding call arcs are also drawn.

   If the *a.out* name, that is, the program name in the configuration file, is different from the *a.out* name in the current display, a confirmation dialog appears to allow you to decide whether or not you still wish to load the file.
3. If after loading a configuration file, you wish to return the function call tree back to its previous state, select the *Filter* menu, and then the *Undo* option.

Looking at source code

Xprofiler provides several ways for you to view your source code. You can view the source or disassembler code for your application on a per-function basis. This also applies to any included function code your application may use.

When you view source or included function code, you use the *Source Code* window. When you view disassembler code, you use the *Disassembler Code*

window. You can access these windows through the Report menu of the Xprofiler GUI or the Function menu of the function in which you are interested.

Viewing source code

Both the Function menu and Report menu provide the means for you to access the *Source Code* window, from which you will view your code.

To access the *Source Code* window via the Function menu:

1. Click on the function box you are interested in with the right mouse button. The Function menu appears.
2. From the Function menu, select the *Show Source Code* option. The *Source Code* window appears.

To access the *Source Code* window via the Report menu:

1. Select the Report menu, and then the *Flat Profile* option. The *Flat Profile* window appears.
2. From the *Flat Profile* window, select the function you would like to view by clicking on its entry in the window. The entry highlights to show that it is selected.
3. Select the *Code Display* menu, and then the *Show Source Code* option. The *Source Code* window appears, containing the source code for the function you selected.

Using the Source Code window: The Source Code window shows you only the source code file for the function you specified from the Flat Profile window or the Function menu. The Source Code Window looks similar to this:

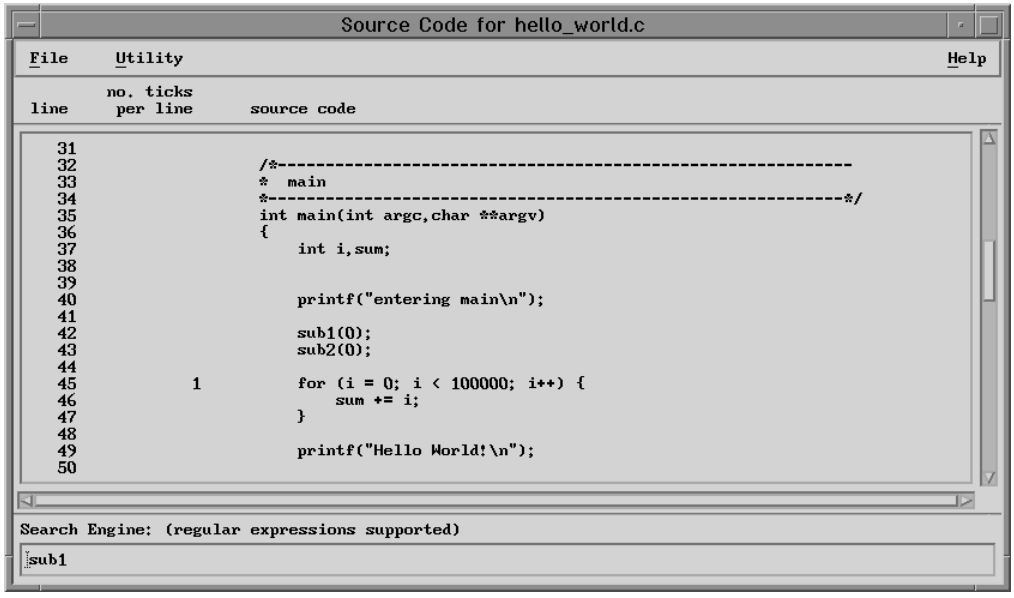


Figure 33. Sample Source Code Window

The Source Code Window contains information in the following fields:

- line

  The source code line number.
- no. ticks per line

Each tick represents .01 seconds of CPU time used. The number that appears in this field represents the number of ticks used by the corresponding line of code. For instance, if the number 3 appeared in this field, for a source statement, this source statement would have used .03 seconds of CPU time. Note that the CPU usage data only appears in this field if you used the **-g** option when you compiled your application. Otherwise, this field is blank.

- source code

  The application's source code.

The *Search Engine* field, at the bottom of the Source Code window, lets you search for a specific string in your source code. For information on using the Search Engine field, see "Using the search engine" on page 54

The Source Code window contains the following menus:

- File

  The *Save As* option lets you save the annotated source code to a file. When you select this option, the Save File Dialog window appears. For more information on using the Save File Dialog window, see "Using the save dialog windows" on page 54

  Select *Close* if you wish to close the Source Code window.

- Utility

  The *Utility* menu contains only one option; *Show Included Functions*.

For C++ users, the *Show Included Functions* option lets you view the source code of included function files that are included by the application's source code.

If a selected function does not have an included function file associated with it or does not have the function file information available because the **-g** option was not used for compiling, the *Utility* menu will be greyed out. The availability of the *Utility* menu serves as an indication of whether or not there is any included function file information associated with the selected function.

When you select the *Show Included Functions* option, the *Included Functions Dialog* window appears, which lists all of the included function files. Specify a file by either clicking on one of the entries in the list with the left mouse button, or by typing the file name in the **Selection** field. Then click on the **OK** or **Apply** button. After selecting a file from the *Included Functions Dialog* window, the *Included Function File* window appears, displaying the source code for the file that you specified.

Viewing disassembler code

Both the Function menu and Report menu provide the means for you to access the *Disassembler Code* window, from which you can view your code.

To access the *Disassembler Code* window via the Function menu:

1. Click on the function you are interested in with the right mouse button. The Function menu appears.
2. From the Function menu, select the *Show Disassembler Code* option. The *Disassembler Code* window appears.

To access the *Disassembler Code* window via the Report menu:

1. Select the Report menu, and then the *Flat Profile* option. The *Flat Profile* window appears.

2. From the *Flat Profile* window, select the function you would like to view by clicking on its entry in the window. The entry highlights to show that it is selected.
3. Select the *Code Display* menu, and then the *Show Disassembler Code* option. The *Disassembler Code* window appears, and contains the disassembler code for the function you selected.

Using the Disassembler Code window: The *Disassembler Code* window shows you only the disassembler code for the function you specified from the Flat Profile window. The Disassembler Code Window looks similar to this:

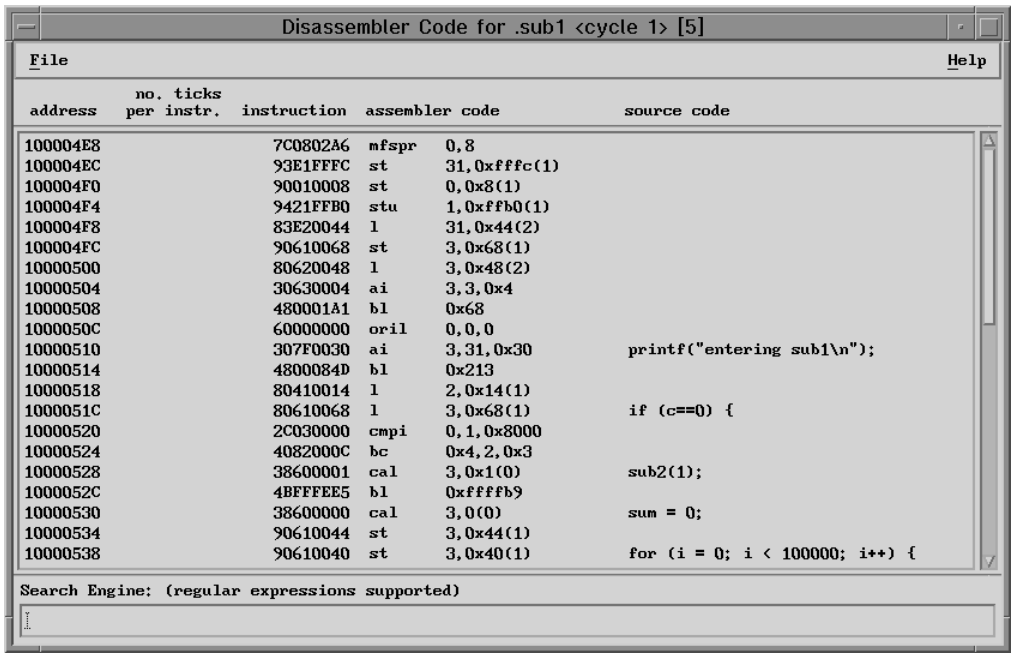


Figure 34. Sample Disassembler Code Window

The Disassembler Code window contains information in the following fields:

- address

  The address of each instruction in the function you selected (from either the Flat Profile window or the function call tree).
- no. ticks per instr.

  Each tick represents .01 seconds of CPU time used. The number that appears in this field represents the number of ticks used by the corresponding instruction. For instance, if the number 3 appeared in this field, this instruction would have used .03 seconds of CPU time.
- instruction

  The execution instruction.
- assembler code

  The execution instruction's corresponding assembler code.
- source code

  The line in your application's source code that corresponds to the execution instruction and assembler code. In order for information to appear in this field, you must have compiled your application with the *-g* compile option.

The *Search Engine* field, at the bottom of the Disassembler Code window, lets you search for a specific string in your disassembler code. For information on using the Search Engine field, see “Using the search engine” on page 54.

The Disassembler Code window contains only one menu:

- File

  Select *Save As* to save the annotated disassembler code to a file. When you select this option, the Save File Dialog window appears. For information on using the Save File Dialog window, see “Using the save dialog windows” on page 54.

  Select *Close* if you wish to close the Disassembler window.

Saving screen images of profiled data

The File menu of the Xprofiler GUI includes an option called *Screen Dump* that lets you capture an image of the Xprofiler main window. This option is useful if you want to save a copy of the graphical display to refer to later. You can either save the image as a file in PostScript format, or send it directly to a printer.

To capture a window image:

1. Select the *File→Screen Dump* options. The Screen Dump menu opens.
2. From the Screen Dump menu, select the *Set Option* option. The *Screen Dump Options Dialog* window appears.

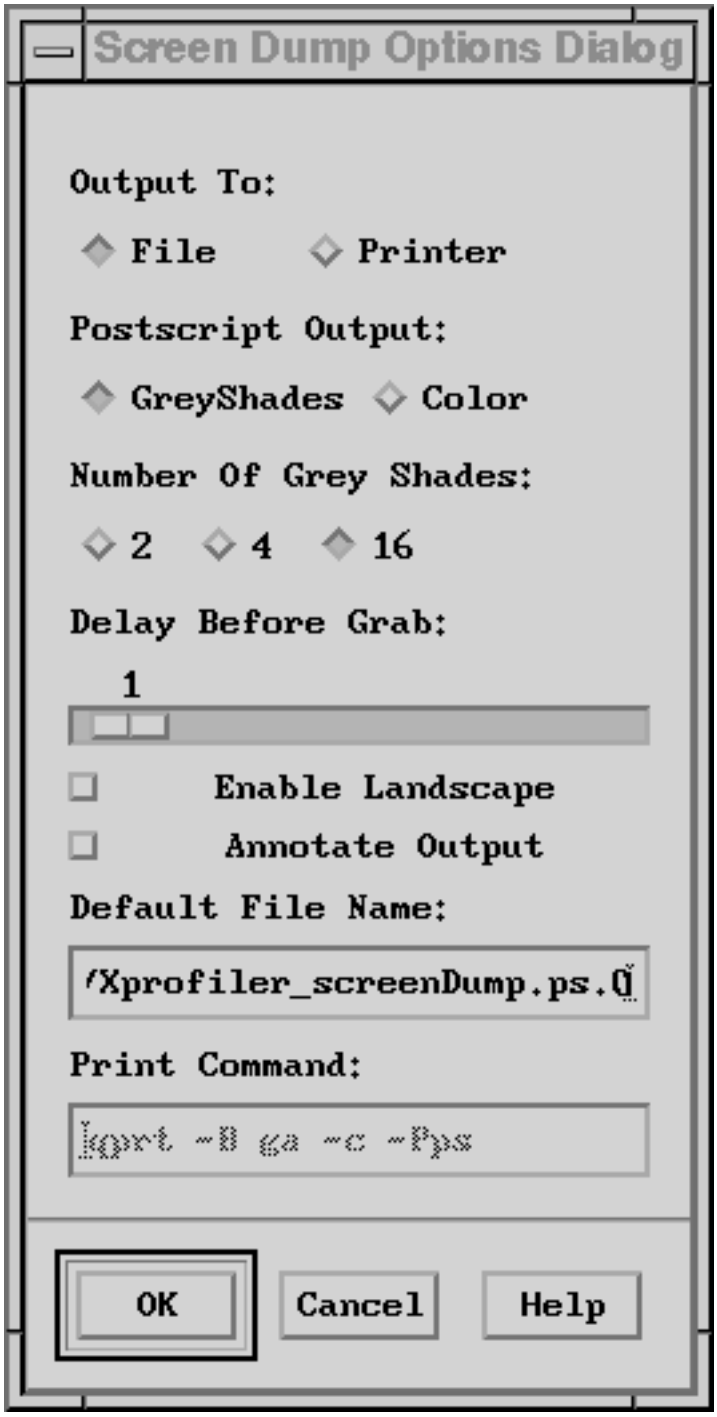


Figure 35. Screen Dump Options Dialog Window

3. Make the appropriate selections in the fields of the *Screen Dump Options Dialog Window* as follows:

– Output To:

  This option lets you specify whether you want to save the captured image as a PostScript file or send it directly to a printer.

  If you would like to save the image to a file, select the *File* button. This file, by default, is named *Xprofiler.screenDump.ps.0*, and is displayed in the

Default File Name field of this dialog window. When you select the *File* button, the text in the *Print Command* field greys out.

If you would like to send the image directly to a printer, select the *Printer* button. The image is sent to the printer you specify in the *Print Command* field of this dialog window. Note that when you specify the Print option, a file of the image is not saved. Also, selecting this option causes the text in the *Default File Name* field to grey out.

– PostScript Output:

  This option lets you specify whether you want to capture the image in shades of grey or in color.

  If you want to capture the image in shades of grey, select the *GreyShades* button. You must also select the number of shades you want the image to include with the *Number of Grey Shades* option, as discussed below.

  If you want to capture the image in color, select the *Color* button. *GreyShades*.

– Number of Grey Shades

  This option lets you specify the number of grey shades that the captured image will include. Select either the 2, 4, or 16 buttons, depending on the number of shades you want to use. Typically, the more shades you use, the longer it will take to print the image.

– Delay Before Grab

  This option lets you specify how long of a delay will occur between activating the capturing mechanism and when the image is actually captured. By default, the delay is set to one second, but you may need time to arrange the window the way you want it. Setting the delay to a longer interval gives you some extra time to do this. You set the delay with the slider bar of this field. The number above the slider indicates the time interval in seconds. You can set the delay to a maximum of thirty seconds.

  To set the delay, place the mouse cursor over the slider. Next, press and hold the left mouse button while moving the slider to the right. When the slider is at the desired number, release the mouse button.

– Enable Landscape (button)

  This option lets you specify that you want the output to be in landscape format (the default is portrait). To select landscape format, select the *Enable Landscape* button.

– Annotate Output (button)

  This option lets you specify that you would like information about how the file was created to be included in the PostScript image file. By default, this information is not included. To do this, select the *Annotate Output* button.

– Default File Name

  If you chose to put your output in a file, this field lets you specify the file name. The default file name is *Xprofiler.screenDump.ps.0*. If you want to change to a different file name, type it over the one that appears in this field.

  If you specify the output file name with an integer suffix (that is, the file name ends with *xxx.nn*, where *nn* is a non-negative integer), the suffix automatically increases by one every time a new output file is written in the same Xprofiler session.

– Print Command

If you chose to send the captured image directly to a printer, this field lets you specify the print command. The default print command is *qprt -B ga -c -Pps*. If you would like to use a different command, type the new command over the one that appears in this field.

Press the **OK** button. The *Screen Dump Options Dialog* window closes.

Once you have set your screen dump options, you need to select the window, or portion of a window, you wish to capture. From the Screen Dump menu, select the *Select Target Window* option. A cursor in the image of a hand appears after the number of seconds you specified. At any time you wish to cancel the capture, you may do so by clicking on the right mouse button. The hand-shaped cursor will change back to normal and the operation will be terminated.

To capture the entire Xprofiler window, place the cursor in the window and then click the left mouse button.

To capture a portion of the Xprofiler window:

1. Place the cursor in the upper left corner of the area you wish to capture.
2. Press and hold the middle mouse button and drag the cursor diagonally downward, until the area you wish to capture is within the rubberband box.
3. Release the middle mouse button to set the location of the rubberband box.
4. Press the left mouse button to capture the image.

If you chose to save the image as a file, the file is stored in the directory you specified. If you chose to print the image, the image is sent to the printer you specified.

Chapter 3. Analyzing program performance using the PE Benchmarker toolset

This chapter describes the tools and utilities of the PE Benchmarker toolset. You can use these tools to collect and analyze program event trace or hardware performance data. Specifically, this chapter describes:

- the Performance Collection Tool (PCT) for collecting MPI traces or hardware/operating system profiles.
- a set of utilities for converting AIX trace records output by the PCT into a format that can be analyzed within third party tools or other utilities that IBM supplies.
- the Profile Visualization Tool (PVT) for analyzing hardware/operating system profiles collected by the PCT.

What is the PE Benchmarker?

The PE Benchmarker is a suite of applications and utilities that you can use to analyze the performance of programs run within the IBM Parallel Environment for AIX. The PE Benchmarker suite consists of:

- **the Performance Collection Tool (PCT)**. This tool enables you to collect either MPI and user event data or hardware and operating system profiles for one or more application processes (or ″tasks″). This tool is built on dynamic instrumentation technology, the *Dynamic Probe Class Library* (*DPCL*). Unlike more traditional tools for collecting message-passing and other performance information, the PCT, because it is built on DPCL, enables you to insert and remove instrumentation probes into the target application while the target application is running. More traditional tools require the application to be instrumented through compilation or linking. This often results in more instrumentation being inserted into the application than is actually needed, and so such tools are more likely to create situations in which the instrumented version of the application is no longer representative of the actual, uninstrumented, version of the application. Since the PCT enables you to make the decision of what data is collected at run time, this typically results in a more acceptable intrusion cost of the instrumentation. What's more, the files output by the PCT are output on each machine running instrumented processes rather than on a single, centralized, machine. This means that your analysis can be efficiently scaled to collect information on a large number of processes running on a large number of nodes.
- **a set of Unified Trace Environment (UTE) utilities**. When you collect MPI and user event traces using the PCT, the collected information is saved, on each machine running instrumented processes, as a standard AIX event trace file. The UTE utilities enable you to convert one or more of these AIX trace files into UTE interval files. While an AIX event trace file has a time stamp indicating the point in time when an event occurred, UTE interval files take this information to also determine how long an event lasts before encountering the next event. Because they include this duration information, UTE interval files are easier to visualize than traditional AIX event trace files. The UTE utilities are:
  - the **uteconvert** utility which converts AIX event trace records into UTE interval trace files.
  - the **utemerge** utility which merges multiple UTE interval files into a single UTE interval file.
  - the **utestats** utility which generates statistics tables from UTE interval files.

  - the **slogmerge** utility which converts and merges UTE interval files into a single SLOG file for analysis within Argonne National Laboratory's Jumpshot tool.
- **the Profile Visualization Tool (PVT)**. When you collect hardware and operating system profiles using the PCT, the collected profile information is saved, on each machine running instrumented processes, as netCDF (network Common Data Form) files. The PVT can read netCDF files and summarize the profile information in reports.

The following figure illustrates how the various tools in the PE Benchmarker toolset work together to enable you to analyze the performance of programs run within the IBM AIX Parallel Environment. Please note that Jumpshot is not part of the PE Benchmarker toolset, but is instead a public domain tool developed at Argonne National Laboratory. It is shown in the figure below, because PE Benchmarker provides the **slogmerge** utility for converting UTE files into the SLOG format required by Jumpshot.

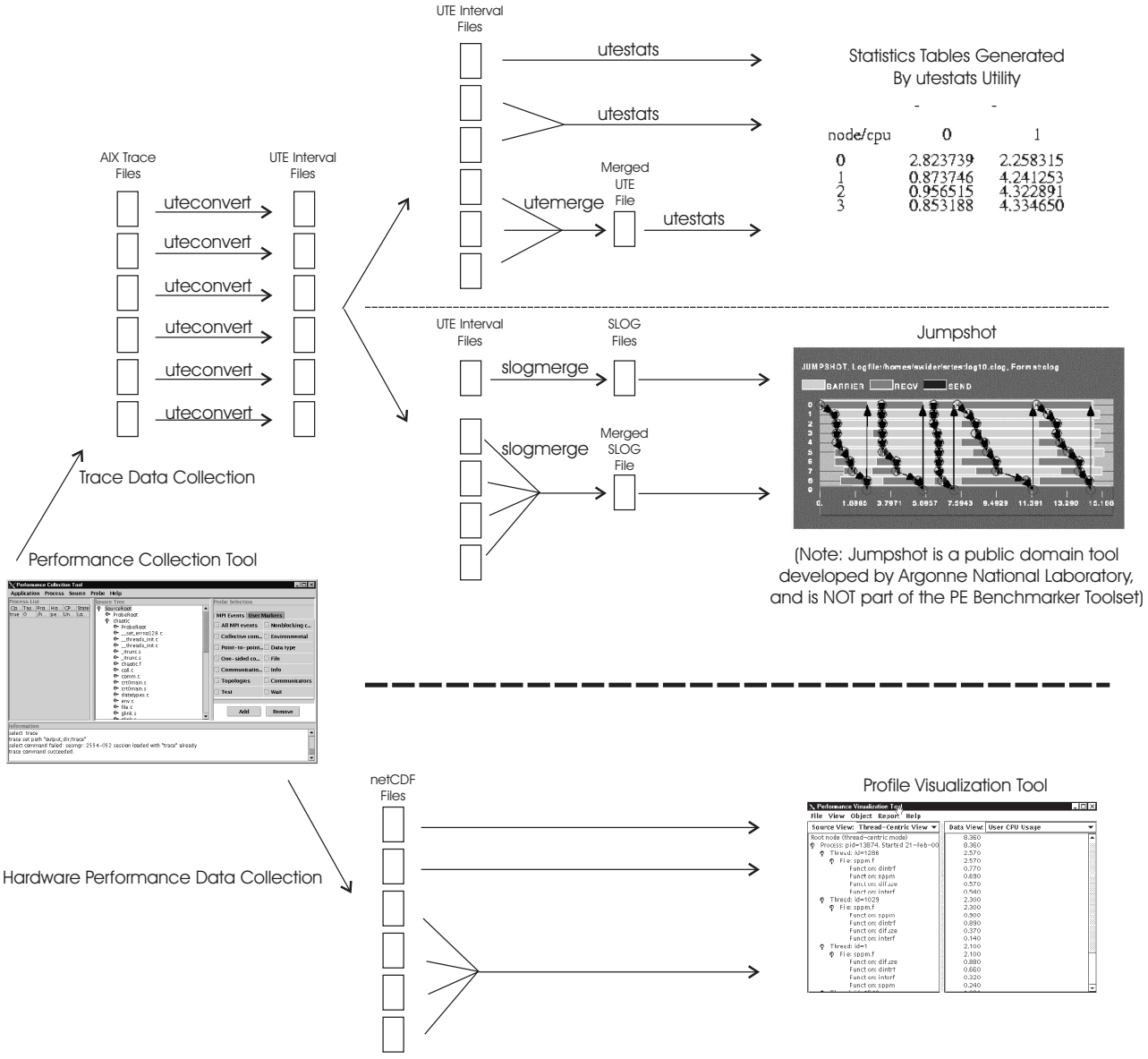


Figure 36. Overview of the PE Benchmarker Toolset

The preceding figure illustrates the procedure for collecting and analyzing data using the PE Benchmarker toolset. This procedure starts with the PCT. When using the PCT, you must select the type of data you are collecting — either MPI and user event trace data or hardware and operating system performance data. You use the PCT to connect to existing processes, or start processes running (which also connects to the processes). By ″connect to processes″ we mean the PCT establishes a communication connection that enables it to control the process' execution (suspend, resume, and terminate the process), and also instrument the process with data collection probes. Data files containing the collected information will be generated on each machine running at least one instrumented process. The format of the files generated depends on the type of data you are collecting.

- If you are collecting MPI and user event trace data, standard AIX trace files will be generated. You will first need to take the AIX trace files output by the PCT

and convert them, using the **uteconvert** utility, into UTE interval files. If you want to view statistical tables of the information contained in the UTE interval files, you can use the **utestats** utility. You can optionally merge multiple UTE files into a single UTE file using the **utemerge** utility before using the **utestats** utility to generate the statistical tables. If you instead want to view the information contained in the UTE interval files graphically, you can convert them into SLOG files which are readable by Argonne National Laboratory's Jumpshot tool. To convert UTE interval files into SLOG files, you use the **slogmerge** utility. The **slogmerge** utility can convert a single UTE interval file into a single SLOG file, or it can convert multiple UTE interval files into a single, merged, SLOG file.

- If you are collecting hardware performance data, netCDF files will be generated. You can use the PVT to generate graphs and reports of the information contained in the netCDF files.

Using the Performance Collection Tool

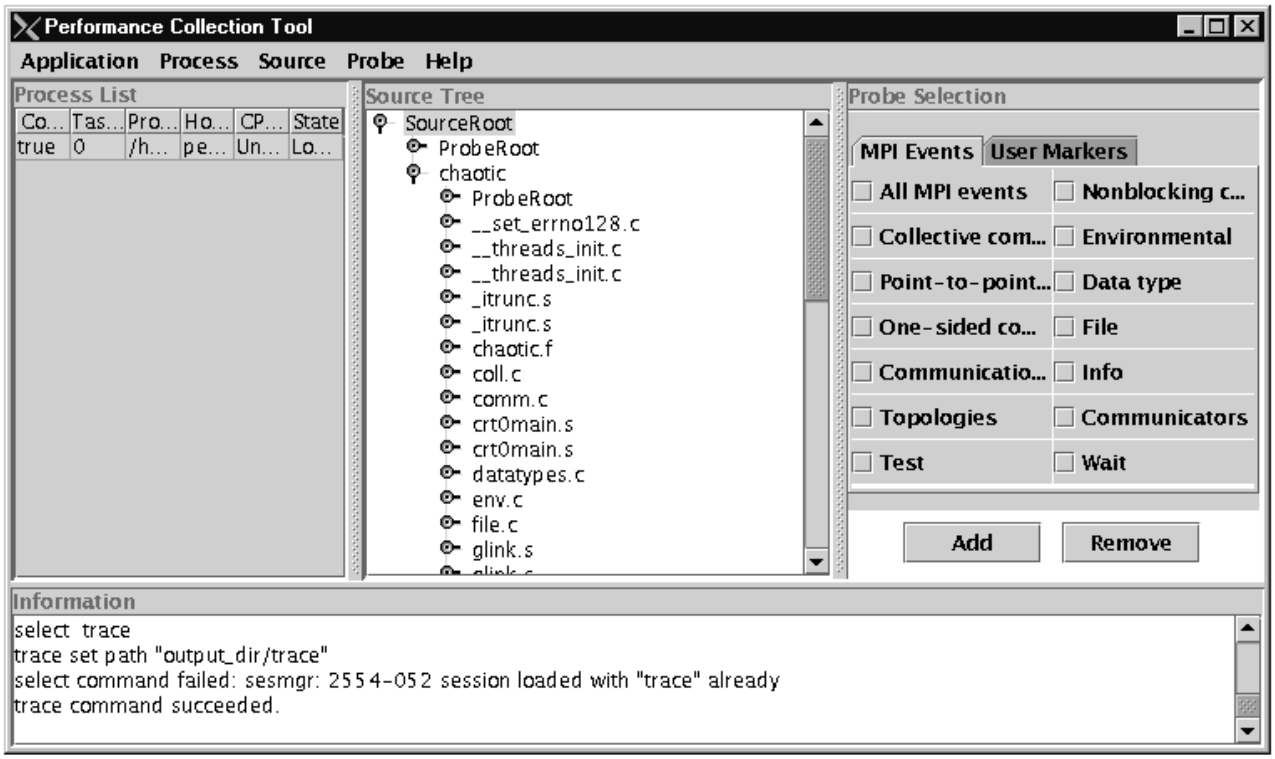
This section describes how to collect MPI and user event traces or hardware and operating system profiles for a particular serial or POE program's run. It describes how you can use the PCT's graphical user interface or command-line interface to:

- connect to a running application, or (if the application you want to examine is not already running) load an application and connect to it.
- select the type of data to collect (either MPI and user event traces, or hardware and operating system profiles).
- start and stop execution of the target application.
- install performance collection probes into the target application to collect the MPI, user event trace, or hardware profile information.
- remove the performance collection probes from the target application when you are through collecting the performance data.
- Disconnect from, or terminate, the target application processes.

For information on the tool's graphical user interface, refer to "Using the Performance Collection Tool's Graphical User Interface". For information on the tool's command-line interface, refer to "Using the Performance Collection Tool's Command-Line Interface" on page 105.

Using the Performance Collection Tool's Graphical User Interface

This section describes how you can use the PCT's graphical user interface to collect either MPI and user event traces, or hardware and operating system profiles. This section begins with a brief overview of the tasks you can perform using the PCT's graphical user interface, and then describes each of these tasks in more detail. You can also operate the PCT using its command-line interface. For information on the tool's command-line interface, refer to "Using the Performance Collection Tool's Command-Line Interface" on page 105.



Here's an overview of the steps you'll follow when using the PCT's graphical user interface to collect either MPI and user event traces, or hardware and operating system profiles. More detailed instructions on each of the tasks summarized are provided later in this chapter. To use the PCT, you:

1. Start the PCT by using the **pct** command. For more information, refer to "Starting the Performance Collection Tool" on page 102.
2. Either load and start a new application, or connect to a running application.
   - To load and start a new application, use the Load Application Dialog to load either a serial or POE application. Using the Load Application Dialog, you can select whether you would like to merely load the application, or load the application and start its execution. If you choose to merely load the application, its execution will be suspended at its first executable instruction. This enables you to install performance collection probes before later starting application execution.
   - To connect to a running application, use the Connect Application Dialog. Using the Connect Application Dialog, you can connect to a serial or POE application. If connecting to a POE application, you can select whether you would like to connect to all processes in the POE application, or just the controlling, ″home node″, POE process. Connecting to only the controlling POE process will enable you to later connect to select tasks in the POE application, and may be desirable for performance reasons.
3. Select the type of data you will be collecting using the PCT. You can collect:
   - MPI and user event traces for analysis using the **utestats** utility or a graphical visualization tool like Jumpshot.
   - hardware and operating system profiles for analysis within the PVT.

4.

If you are collecting:	Then:
MPI and user event traces.	Use the Probe Selection Panel of the PCT's Main Window to specify which MPI events you want to collect data for. For example, you can select ″All MPI events″, ″Collective communication″, ″Point-to-point communication″, and so on. In addition to specifying MPI trace data to be collected, you can also add user markers to processes to mark events or states of interest. Marking these states or events of interest gives you a frame of reference when analyzing the trace record in a graphical visualization tool like Jumpshot. You can also use user markers to mark locations where tracing should be stopped or started. Since you can add MPI probes only at a program, file, or function level (meaning that the entire program, file, or function will be traced), this gives you more control over which part of your program is traced.
hardware and operating system profiles.	Use the Probe Selection Panel of the PCT's Main Window to specify the hardware and operating system information you want to collect for later analysis within the PVT.

5. When you are done collecting data, you can terminate connected processes, disconnect from the processes, and/or exit the PCT.

In addition to the tasks summarized above, you can also:

- display the contents of source files in the View Source window.
- use a search string to locate functions within the Main Window's Source Tree.
- set user preferences. Specifically, you can set the:
  - search path used by the tool to locate source files for display
  - size of the buffers used when creating MPI trace files
  - maximum size of the MPI trace files
  - types of events included in MPI trace files.
- start and stop execution of connected processes. You might, for example, wish to suspend execution of your application prior to instrumenting it, and resume execution after probes have been added.
- examine standard output and error from, and send standard input to, the application using the I/O Console Window.

Starting the Performance Collection Tool

You can start the PCT in either graphical-user-interface mode or command-line mode. For instructions on starting the PCT in command-line mode, refer to “Using the Performance Collection Tool's Command-Line Interface” on page 105. To start the PCT in graphical-user-interface mode:

1. Enter the **pct** command at the AIX command prompt.

   ```
   $ pct
   ```

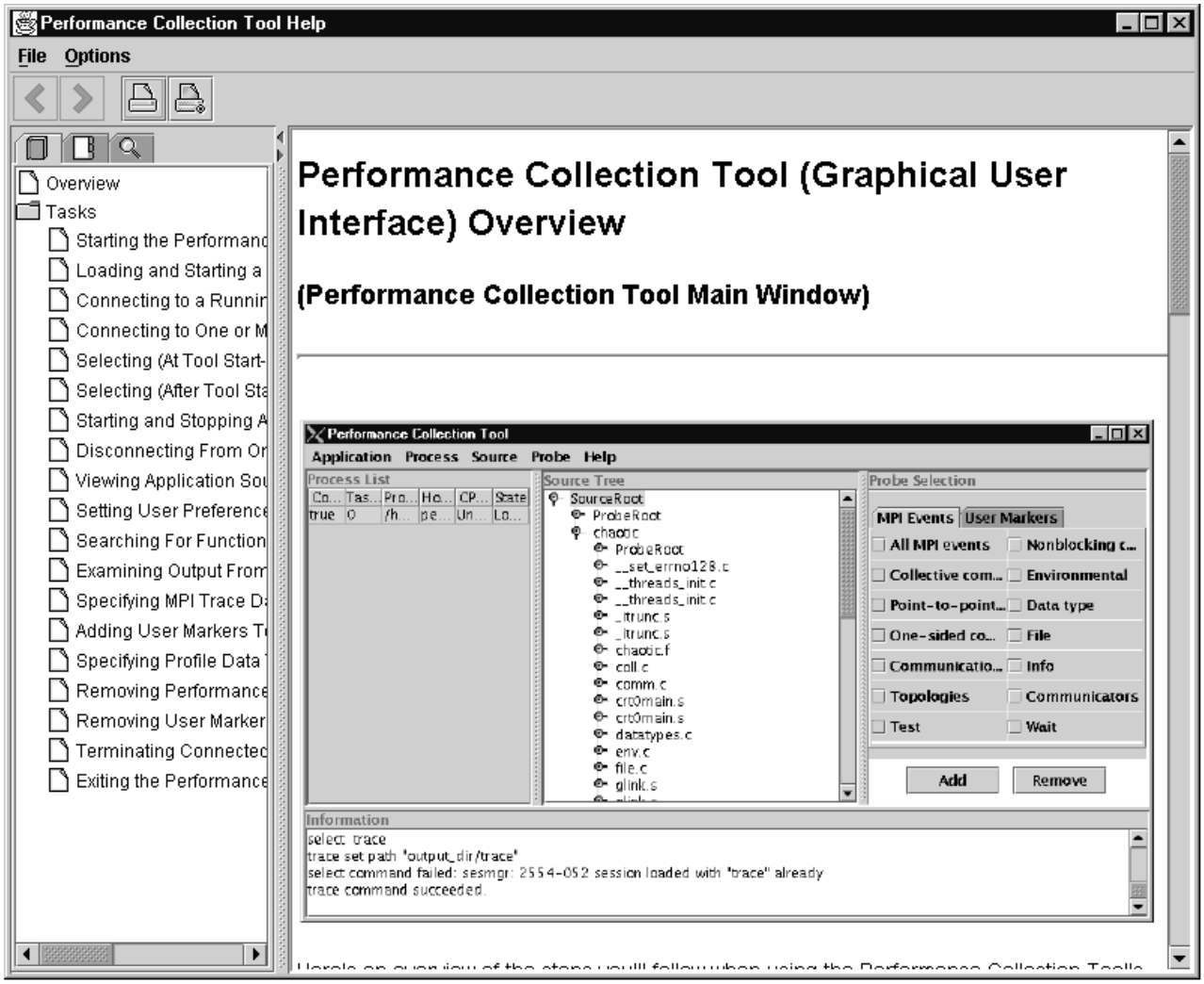
   Doing this starts the PCT in graphical-user-interface mode and opens its first window — the Welcome Dialog.

2. The Welcome Dialog provides option buttons that enable you to select whether you would like to load a new application or connect to an existing one.

If:	Then:
You want to examine an application that is not already running.	Select the **Load a new application** option button and click the **OK** command button. Doing this closes the Welcome Dialog, and opens the Load Application Dialog. The Load Application Dialog will enable you to specify the serial or POE program you wish to run.
You want to examine an application that is already running.	Select the **Connect to a running application** option button and click the **OK** command button. Doing this closes the Welcome Dialog, and opens the Connect Application Dialog. The Connect Application Dialog will enable you to specify the serial or POE program to which you want to connect.
You do not want to make the decision between whether to load a new, or connect to an existing, application at this time.	Click on the **Cancel** command button. Doing this closes the Welcome Dialog and opens the PCT's Main Window. Since you have neither loaded a new application, nor connected to an existing application, the Main Window will not provide any application information at this time.

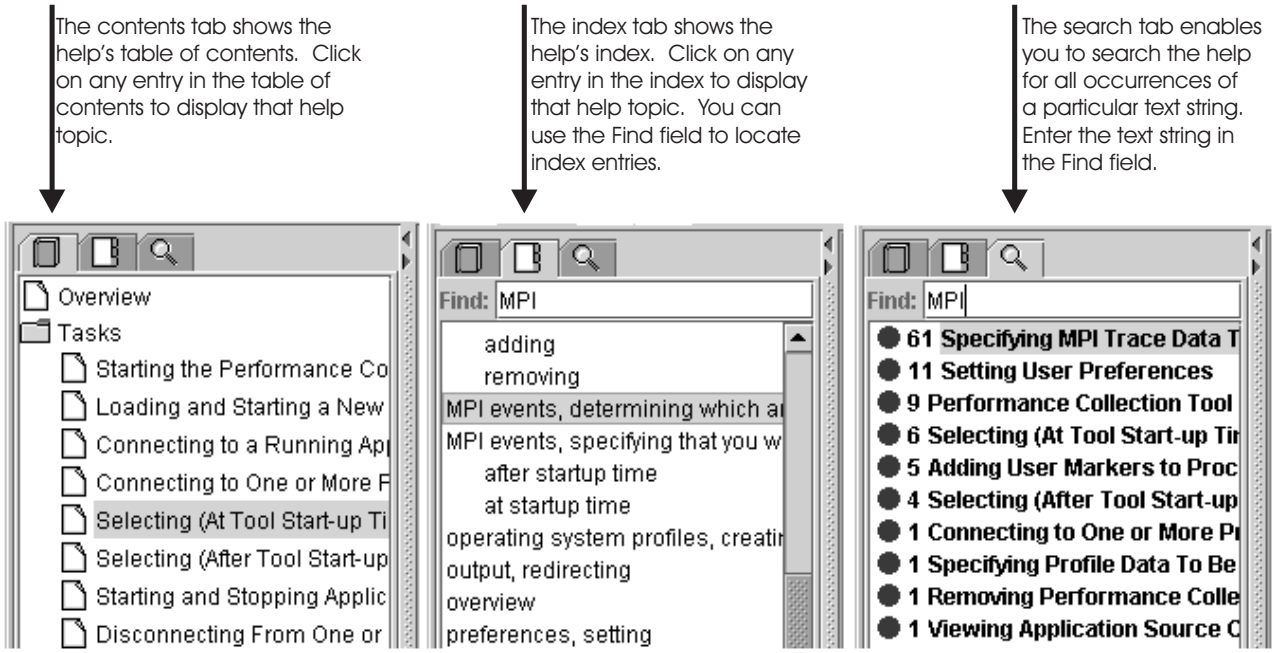
Accessing the Performance Collection Tool's online help system

The PCT's graphical user interface has been designed to be intuitive and easy to use. If you do have any trouble using it to accomplish the tasks outlined in "Performance Collection Tool (Graphical User Interface) Overview" on page 101, refer to the PCT's online help system. To access the tool's online help, select **Help → Contents** off the main window's menu bar, or else press the **Help** button that appears on many of the PCT's dialogs. Doing this opens the PCT help window.



If you open the help from one of the PCT's dialogs, a help topic describing that dialog is displayed. If you open the help from the main window, a task overview topic is displayed.

The PCT help contains topics for each of the major tasks you can perform with the PCT. The left hand pane of the window enables you to navigate the help system to display the needed help topic in the right hand pane. There are three ways to navigate the help system — using the contents tab, using the index tab, or using the search tab:



- the contents tab is displayed by default. Simply click on any entry in the contents tab to display the help topic.
- the index tab shows an index of the entire help system. Simply click on any entry in the index to display its associated help topic. To search the index, type a string in the **Find** field and press **<enter>**. The first index entry containing the string is highlighted. Press **<enter>** again to search for the next occurrence of the string in the index.
- the search tab enables you to search the help for all occurrences of a text string. Simply type the string in the **Find** field and press **<enter>**. A list of all help topics containing the string is displayed. The topics are listed in descending order according to the number of occurrences of the string. The help topic with the most occurrences of the string is displayed by default.

Using the Performance Collection Tool's Command-Line Interface

This section describes how you can use the PCT in command-line mode to collect either MPI and user event traces or hardware and operating system profiles. The purpose of this section is to illustrate how the various subcommands of the **pct** command can be used to instrument serial or POE programs. Note, however, that this section does not necessarily describe all the options of all the **pct** subcommands. For complete reference information on any of the subcommands described in this section, refer to the **pct** command's man page in "Appendix A. Parallel environment tools commands" on page 139.

This section begins with a brief overview of the tasks you can perform using the PCT's command-line interface, and then describes each of these tasks in more detail. You can also operate the PCT using its graphical user interface. For information on how to do this, refer to "Using the Performance Collection Tool's Graphical User Interface" on page 100.

Performance Collection Tool (Command-Line Interface) Overview

To use the PCT's command-line interface to collect either MPI and user event traces or hardware and operating system profiles:

1. Start the PCT in command-line mode by issuing the **pct** command with its **-c** option. You can optionally specify the **-s** option to instruct the PCT to read its subcommands from a script file. For more information, refer to “Starting the Performance Collection Tool In Command-Line Mode” on page 107.
2. Either load and start a new application, or connect to a running application.
   - To load and start a new application, use the **load** subcommand to load either a serial or POE application. When you load an application, its process execution will be suspended at its first executable instruction. To start execution of one or more loaded application processes, issue the **start** subcommand. For more information, refer to “Loading and Starting a New Application” on page 110.
   - To connect to a running application, use the **connect** subcommand. You can connect to a serial process or a POE home node process using this subcommand. Once connected to a POE home node process, you can issue the **connect** subcommand again to connect to one or more of its individual tasks. For more information, refer to “Connecting to a Running Application” on page 111.

   When you load or connect to a serial or POE application, two task groups are created. A *task group* is simply a named set of tasks — in this case, the task groups are named ″*all*″ and ″*connected*″. Task groups are intended for when you are working with POE applications as opposed to serial applications. The *all* task group represents all the tasks in the POE application, while the *connected* task group represents the POE application’s connected tasks only. You can also create your own named task groups. Task groups enable you to more easily manipulate the tasks of a POE application, since many of the PCT’s subcommands are designed to operate upon one or more tasks. By default, the tasks operated upon are those in a ″current task group″ that you specify. By default, the current task group is the automatically-created task group *connected*. If you are instrumenting a serial application, you naturally do not need to concern yourself with task groups. You should be aware, however, that the *all* and *connected* groups are still created by the PCT. For more information on task groups, refer to “Grouping Tasks of a POE Application” on page 108.
3. Select the type of data you will be collecting using the PCT. You can collect either:
   - MPI and user event traces for analysis using the **utestats** utility or a graphical visualization tool like Jumpshot.
   - hardware and operating system profiles for analysis within the PVT.

   To specify which type of data you’ll be collecting, use the **select** subcommand. For more information, refer to “Selecting the Type of Probe Data To Be Collected” on page 115.
4. Set an output location for files output by the PCT, and add probes to collect data.

If:	Then:
you are collecting MPI and user event traces.	a. Set the output location for the trace files that are generated by the PCT. To do this, use the **trace set** subcommand. For more information, refer to “Setting the Output Location and Other Preferences for the AIX Trace Files Generated” on page 116. b. Add MPI trace probes and/or custom user markers using the **trace add** subcommand. For more information, refer to “Adding MPI Trace Probes to Processes” on page 117 and “Adding User Markers to Processes” on page 119. When you are done collecting the trace data, you can remove the probes using the **trace remove** subcommand. For more information, refer to “Removing MPI Trace Probes From Processes” on page 119 and “Removing User Markers From Processes” on page 120.
you are collecting hardware and operating system profile information.	a. Set the output location for the profile files that are generated by the PCT. To do this, use the **profile set path** subcommand. For more information, refer to “Setting the Output Location for the netCDF Files Generated” on page 121. b. Add the profile probes to processes using the **profile add** subcommand. For more information, refer to “Adding Hardware Profile Probes to Processes” on page 121. When you are done collecting the profile data, you can remove the probes using the **profile remove** subcommand. For more information, refer to “Removing Hardware Profile Probes From Processes” on page 123.

5. When you are done collecting data, you can terminate connected processes using the **destroy** subcommand, or disconnect from the processes using the **disconnect** subcommand. To exit the PCT, issue the **exit** subcommand. For more information, refer to “Terminating Connected Processes” on page 124, “Disconnecting From the Application” on page 124, and “Exiting the Performance Collection Tool” on page 125.

In addition to the tasks summarized above, you can also:

- suspend and resume execution of connected processes by issuing the **suspend** and **resume** subcommands. You might, for example, wish to suspend execution of your application prior to instrumenting it, and resume execution after the probes have been added. For more information, refer to “Suspending and Resuming Application Execution” on page 112.
- send standard input text to your application using the **stdin** subcommand. For more information, refer to “Sending Standard Input Text to the Application” on page 113.
- Display the contents of source files using the **list** subcommand. For more information, refer to “Displaying the Contents of a Source File” on page 113.

Starting the Performance Collection Tool In Command-Line Mode

To start the PCT in command-line mode, enter, at the AIX command prompt, the **pct** command with its **-c** option:

```
pct -c
```

The PCT displays the `pct>` command prompt. You can now enter PCT subcommands at this prompt.

When starting the PCT in command-line mode, you can optionally specify the **-s** option to instruct the PCT to read subcommands from a particular script file of PCT subcommands. For example, to have the PCT read the subcommands in the script file *myscript.cmd*:

```
pct -c -s myscript.cmd
```

For more information on PCT script files, refer to "Creating and Running PCT Script Files" on page 125.

The first thing you'll want to do after starting the PCT is either connect to a running application, or load and connect to a new application. If the application you wish to examine is already running, you can connect to it; refer to "Connecting to a Running Application" on page 111. If the application you wish to examine is not already running, you can load it; refer to "Loading and Starting a New Application" on page 110. If you are going to connect or load a POE application, you need to understand the concept of task groups; refer to "Grouping Tasks of a POE Application".

Getting help on the PCT's command-line interface

To get a listing of all of the PCT's subcommands, enter the **help** subcommand at the `pct>` prompt.

```
pct> help
```

To get the syntax of a particular subcommand, enter the **help** subcommand followed by the name of the subcommand whose syntax you want displayed. For example, to get the syntax of the **load** subcommand.

```
pct> help load
```

Grouping Tasks of a POE Application

In the Parallel Operating Environment, the multiple cooperating processes of your program are referred to as ″tasks″. Many of the PCT subcommands are designed to operate on one or more tasks of a POE application. By default, the tasks operated upon are those in a ″current task group″ that you can specify. A task group is simply a named set of tasks. Two such task groups — *all* and *connected* — are created automatically when you either connect to a running application (using the **connect** subcommand), or load a new application (using the **load** subcommand). The *all* task group represents all the tasks in the POE application. The *connected* task group is the current task group by default — it represents the POE application's connected tasks only. You can also create your own task groups.

By default, the current task group will be *connected*; the subcommands you issue will act upon all connected tasks in the POE application. You can change the current task group to be the automatically created group *all*, or a task group that you have created. You can also, for all of the subcommands that act upon task groups, specify a set of tasks or a task group when issuing the subcommand. If you do this, the subcommand will operate on the tasks specified rather than the current task group. For example, consider the **suspend** subcommand for suspending execution of one or more tasks. If you issue this subcommand without options as in:

```
pct> suspend
```

The tasks in the current task group are suspended. However, if you specify a task list using the **task** clause, you suspend execution for the tasks specified — in this next example tasks 0 through 5:

```
pct> suspend task 0:5
```

Note: When using the task clause, the tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

You can also specify a named task group (other than the current task group) using the **group** clause:

```
pct> suspend group workers
```

To understand why you might want to specify a task group, consider the following example. Say that the application you're examining follows the master/workers model in which one task (the ″master″) coordinates the activities of all the other tasks — the ″workers″. You could create two task groups — one containing just the master task, and the other containing all the other tasks. To do this, you would use the **group** subcommand with its **add** clause. To create a task group *master* containing just task 0:

```
pct> group add master 0
```

To create a task group *workers* containing the tasks 1 through 10:

```
pct> group add workers 1:10
```

Once these groups are created, you can make either one the current task group. To do this, you would use the **group** subcommand with its **default** clause. For example, the following subcommand sets the current task group to be the task group *master*:

```
pct> group default master
```

While *master* is the current task group, any subcommands that operate upon tasks will operate only upon task 0 — the only task in the group *master*. To make the group *workers* the current task group:

```
pct> group default workers
```

While you cannot modify or delete the two groups that the PCT automatically creates (*all* and *connected*), you can modify and delete the groups that you have created. To add tasks 11 though 20 to the task group *workers*:

```
pct> group add workers 11:20
```

To delete task 11 from the task group *workers*:

```
pct> group delete workers 11
```

To delete the entire task group *workers*:

```
pct> group delete workers
```

Notes:

1. If you are instrumenting a serial application, you naturally do not need to concern yourself with task groups. You should be aware, however, that the *all* and *connected* groups are still created by the PCT.
2. You can list the existing task groups, or the members of a particular task group, using the **show** subcommand. For example, the following subcommand lists the existing task groups:

```
pct> show groups
Default     Group Name
-------     ----------
            all
@           connected
pct>
```

The @ symbol indicates which group is the current task group.

To list the tasks in the task group *all*:

```
pct> show group all
Tid Program Name             Host              Cpu Type State
--- ------------------------ ----------------- -------- ------
0   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
1   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
2   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
3   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
pct>
```

Loading and Starting a New Application

If the serial or POE application you wish to examine is not already running, you can load it onto one or more nodes. When you load an application using the **load** subcommand, it is loaded in a stopped state with execution suspended at the first executable instruction. You can then start its execution using the **start** subcommand.

To load a serial application, you simply supply the **load** subcommand with the absolute path to the executable. The **exec** clause indicates the absolute path to the executable. If the application takes arguments, you can specify them using the **args** clause. For example:

```
pct> load exec /u/example/bin/foo args "a b c"
```

If loading a POE application, you specify the **poe** clause, and can also supply any POE arguments using the **poeargs** clause. For information on the POE command-line flags available to you, refer to the manual *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

The procedure for loading a POE application differs depending on whether the application follows the Single Program Multiple Data (SPMD) or Multiple Program Multiple Data (MPMD) model. If your program follows the SPMD model, you specify the absolute path to the executable using the **exec** clause:

```
pct> load poe exec /u/example/bin/parallel_foo poeargs "-procs 4 -hfile /tmp/host.list"
```

If your program follows the MPMD model, you supply the absolute path to a POE commands file (which lists the individual programs to load) using the **mpmdcmd** clause:

```
pct> load poe mpmdcmd /u/example/bin/foo.cmds poeargs "-procs 3 -hfile /tmp/host.list"
```

For information on creating a POE commands file for loading multiple programs, refer to the manual *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

The **load** subcommand also enables you to specify that standard input, standard output, or standard error should be redirected. To read standard input from a file, use the **stdin** clause:

```
pct> load exec /u/example/bin/foo args "a b c" stdin input_file
```

To redirect standard output to a file, use the **stdout** clause:

```
pct> load exec /u/example/bin/foo args "a b c" stdout output_file
```

To redirect standard error to a file, use the **stderr** clause:

```
pct> load exec /u/example/bin/foo args "a b c" stderr error_file
```

When you load an application, two task groups — *all* and *connected* — are automatically created, and *connected* is made the current task group. Task groups are important to know about only if you are working with a POE application and are described in “Grouping Tasks of a POE Application” on page 108. Also note that the application is loaded in a stopped state with execution suspended at the first executable instruction. To start execution of the application, use the **start** subcommand:

```
pct> start
```

Connecting to a Running Application

If the serial or POE application you wish to examine is already running, you can connect to it using the **connect** subcommand. To list the processes to which you can connect, use the **show** subcommand with its **ps** clause:

```
pct> show ps
Pid   Command
----- ----------------------------------------------------------------
10652 /home/strofino/dpcltest/WORK/prod_cons
13256 /etc/dpcld /tmp/dpclsd
13316 /home/strofino/dpcltest/WORK/prod_cons
14302 /usr/lpp/ppe.dpcl/dpcl_beta/bin/poe
18108 /home/strofino/dpcltest/WORK/prod_cons
20614 /u/alfeng/public/perf/seqsleep
21996 /u/alfeng/bin/sesmgr
22644 /home/strofino/dpcltest/WORK/prod_cons
22802 java com/ibm/ppe/perf/main/Startup -l /u/alfeng/bin/sesmgr -cmd
23236 -ksh
24894 /etc/dpcld /tmp/dpclsd
27632 -ksh
pct>
```

If you are connecting to a serial application, you simply supply the process ID of the process you wish to connect to using the **pid** clause of the **connect** subcommand.

```
pct> connect pid 12345
```

If you are connecting to a POE application, you connect to the processes in two steps. First, you issue the **connect** subcommand to connect to the controlling, home node, POE process. Once connected to the controlling POE process, you can then reissue the **connect** subcommand to connect to any of its processes. For example, to connect to the application whose AIX process ID is 12345:

```
pct> connect poe pid 12345
```

When you connect to the POE home node process, the PCT creates two task groups — *all* and *connected*. The *all* task group refers to all of the tasks in the application, while the *connected* task group refers only to connected tasks. The *connected* task group will initially be empty since no tasks are connected. You can list the existing task groups by issuing the **show** subcommand with its **groups** clause:

```
pct> show groups
Default     Group Name
-------     ----------
            all
@           connected
pct>
```

To connect to all tasks in the POE application:

```
pct> connect group all
```

To connect to select tasks in the POE application, use the **task** clause:

```
pct> connect task 2,3
```

Suspending and Resuming Application Execution

The PCT enables you to suspend and resume execution of connected processes by issuing the **suspend** and **resume** subcommands. You might, for example, wish to suspend execution of your target application prior to instrumenting it as described in "Collecting MPI Trace and Custom User Marker Information" on page 115. Once your performance collection probes have been added to the application, you could resume the application's execution. By default, the **suspend** and **resume** subcommands act upon the current task group. Unless you have specified another task group to be the current task group, the current task group will be the task group *connected*. The task group *connected* is created automatically by the PCT when you either connect to or load an application (as described in "Connecting to a Running Application" on page 111 and "Loading and Starting a New Application" on page 110). The task group *connected* consists of all connected tasks in a POE application. If you are instrumenting a serial application, you do not need to concern yourself with task groups. If you are instrumenting a POE application, however, it is useful to understand the concept of task groups as described in "Grouping Tasks of a POE Application" on page 108.

To suspend execution of the tasks in the current task group:

```
pct> suspend
```

To suspend execution of tasks in a specific task group (in this case, the task group *connected*), use the **group** clause on the **suspend** subcommand:

```
pct> suspend group connected
```

To suspend a specific set of tasks in a POE application, use the **task** clause on the **suspend** subcommand. To determine how many tasks are available, you can use the **show group** subcommand to list the tasks in the task group *all*:

```
pct> show group all
Tid Program Name             Host             Cpu Type State
--- ------------------------ ---------------- -------- ------
0   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
1   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
2   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
3   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
pct> suspend task 1,3
```

The **resume** subcommand works in the same way. By default, it operates on the current task group:

```
pct> resume
```

But you can override this by specifying a task group:

```
pct> resume group connected
```

or supplying a task list:

```
pct> resume task 1,5
```

Sending Standard Input Text to the Application

If you have loaded an application (as described in “Loading and Starting a New Application” on page 110), you can use the **stdin** subcommand to send standard input text to your application. However, if you have instead merely connected to an application (as described in “Connecting to a Running Application” on page 111), you cannot send standard input text to the application using the **stdin** subcommand.

If you are instrumenting a serial application, the standard input text will be sent to that application process. If you are instrumenting a POE application, the standard input text will be sent to the controlling, ″home node″, POE process. As described in “Loading and Starting a New Application” on page 110, you can, when loading an application using the **load** subcommand, specify that standard input should be read from a file. If you are reading standard input from a file, you cannot use the **stdin** subcommand.

To send a standard input string to the application, specify the string on the **stdin** subcommand. The string must be enclosed in double quotes:

```
> stdin "Now is the time for all good men"
```

If desired, you can use embedded formatting characters (such as **\n**) in your standard input string:

```
> stdin "Now is the time \nfor all good men"
```

To send a newline character to the input stream reading this input data, issue the **stdin** command without any text string:

```
stdin
```

To send an end-of-file character to the input stream reading this input data, use the **eof** clause on the **stdin** subcommand:

```
> stdin eof
```

Displaying the Contents of a Source File

Using the **list** subcommand, you can display the contents of source files. Unless you are certain of the file name of the source file you want to examine, you may want to list the available source files using the **file** subcommand. The **file** subcommand lists, for one or more connected tasks, the associated source file names that match a regular expression you supply. By default, the **file** subcommand acts upon the current task group. Unless you have specified another task group to be the current task group (as described in “Grouping Tasks of a POE Application” on page 108), the current task group will be the task group *connected*. The task group *connected* is created automatically by the PCT when you either connect to or load an application (as described in “Connecting to a Running Application” on page 111 and “Loading and Starting a New Application” on page 110). The task group *connected* consists of all connected tasks in a POE application. If you are instrumenting a serial application, you do not need to concern yourself with task groups. If you are instrumenting a POE application, however, it is useful to understand the concept of task groups as described in “Grouping Tasks of a POE Application” on page 108.

You supply the **file** subcommand with an AIX regular expression file-matching pattern (enclosed in double quotation marks) to match the source files you want to list. For example, to list all the available source files in the current task group:

```
pct> file "*"
Tid     File Id     File Name    Path
---     -------     ---------    -------------
0       0           bar.c        ../../lib/src
0       1           foo1.c       ../../lib/src
0       2           foo2.c       ../src
pct>
```

Although this subcommand, by default, acts upon the current task group, you can specify that it should instead act upon a different task group, or all the tasks in a task list that you supply. This is done by using the **task** or **group** clause on the **file** subcommand. For more information on the **task** and **group** clauses, refer to "Grouping Tasks of a POE Application" on page 108.

After issuing the **file** subcommand, you'll have both the file name and the file identifier of the source file(s) you want to examine. Now you can use the **list** subcommand to display the contents of one or more files. Like the **file** subcommand, the **list** subcommand will, by default, act upon the current task group. Using either the **file** or **fileid** clause of the **list** subcommand, you indicate the file(s) whose contents you want listed.

When listing the contents of files using the **list** subcommand, the PCT uses a special source path to locate the source files. This source path is, by default, the directory in which the PCT was started, and can be displayed using the **sourcepath** clause on the **show** subcommand as in:

```
pct> show sourcepath
Path
----
./
pct>
```

To modify the source path so that the PCT can locate source files that are not located in the directory in which the tool was started, use the **set** subcommand. As with setting your AIX **PATH** environment variable, you separate the various directories in your source path using colons. For example:

```
pct> set sourcepath "/afs/aix/u/jbrady:/afs/aix/u/dlecker"
```

Using the **file** clause, you supply the **list** subcommand with an AIX regular expression file-matching pattern (enclosed in double quotation marks) to match the source file(s) whose contents you want to list. If desired, you can supply additional regular expressions separated by commas (`file "f*","b*"`). For example, the following subcommand lists the contents of the file *bar.c*:

```
pct> list file "bar.c"
```

While this subcommand lists the contents of the first file found in the application that begins with the letter ″f″:

```
pct> list file "f*"
```

Using the **fileid** clause, you identify the file whose contents you want to list using the process identifier(s) returned by the **file** subcommand. For example, the following subcommand lists the contents of the file *bar.c* (whose file identifier is 0):

```
pct> list fileid 0
```

You can also use the **line** clause of the **list** subcommand to list only a portion of the file's contents. Use a colon to separate the ends of the line number range. For example, the following subcommand lists lines 1 through 20 of the file *bar.c*.

```
pct> list file "bar.c" line 1:20
```

To list the next few lines in *bar.c*, simply specify the **next** clause on the **list** subcommand.

```
pct> list next
```

Selecting the Type of Probe Data To Be Collected

The PCT is capable of collecting two different types of information. It can collect:

- MPI and user event traces for analysis using the **utestats** utility or a graphical visualization tool like Jumpshot (a public domain tool developed at Argonne National Lab). For more information on the **utestats** utility, as well as utilities for converting the AIX trace files created by the PCT into a format readable by **utestats** and Jumpshot, refer to "Creating, Converting, and Viewing Information Contained In, UTE Interval Files" on page 126.
- Hardware and operating system profiles for analysis within the PVT. For more information on the PVT, refer to "Using the Profile Visualization Tool" on page 131.

Be aware that, before you can collect either type of information, you must specify, using the **select** subcommand, which type you are interested in:

If you want to collect:	Then:
MPI and user event traces.	Specify the **trace** clause on the **select** subcommand: `select trace`
Hardware and operating system profiles.	Specify the **profile** clause on the **select** subcommand: `select profile`

Note: You can select the type of data to collect only once per load and connect.

Collecting MPI Trace and Custom User Marker Information

Using the PCT, you can collect MPI and user event traces for:

- analysis using the **utestats** utility
- eventual analysis within a graphical visualization tool like Jumpshot

The trace information collected is stored as an AIX trace file on each node running instrumented processes. After you have generated these AIX trace files, you can convert them into the Unified Trace Environment (UTE) format (using the **uteconvert** utility) for analysis using the **utestats** utility. You can then also convert the UTE files into the SLOG format (using the **slogmerge** utility) for analysis within Jumpshot. For more information on the utilities for converting the AIX trace files output by the PCT into formats readable by the **utestats** utility and Jumpshot, refer to "Creating, Converting, and Viewing Information Contained In, UTE Interval Files" on page 126.

In order to collect MPI trace information, the application to be traced must be linked with the *libute_a* library. To cause this UTE library to be added to the link step, set the **MP_UTE** environment variable to *yes*.

Before you can use any of the MPI trace collection subcommands described in this section, you must first specify that you are collecting MPI trace information rather than hardware profile information. Refer to "Selecting the Type of Probe Data To Be Collected" for more information. Once you have indicated that you'll be collecting MPI and/or user event traces, you can select the output location for the trace files generated by the PCT. To do this, you simply supply an output directory and ″base

name″ (file prefix) for the trace files. Refer to “Setting the Output Location and Other Preferences for the AIX Trace Files Generated” for more information. You can collect information about:

- standard MPI messaging events such as collective communication, point-to-point communication, or one-sided communication. This is done by adding MPI data collecting probes to one or more application task. Refer to “Adding MPI Trace Probes to Processes” on page 117 for more information.
- events of interest (such as program function calls). This is done by installing a simple user marker into one or more application task at an instrumentation point in the code. Instrumentation points are locations in the code (such as function call sites) where it is safe to install probes. A simple marker will appear in the trace record as a single point; its position gives you a frame of reference when analyzing a trace record in a graphical visualization tool like Jumpshot.
- states of interest. This is done by installing beginning and ending state user markers in the code at particular instrumentation points. A state will appear in the trace record as a region and, like the simple markers, gives you a frame of reference when analyzing a trace record in a graphical visualization tool like Jumpshot.

Setting the Output Location and Other Preferences for the AIX Trace Files Generated: The trace information collected by the PCT is stored as a separate AIX trace file on each node running instrumented processes. You can select the output location and other preferences for the trace files using the **trace set** subcommand.

To specify:	Use this clause of the trace set subcommand:	For example:
The output location and a ″base name″ prefix for the generated files.	**path**	`pct> trace set path "/home/timf/trace files/mytrace"` Specifies */home/timf/tracefiles* as the location for the generated files. The basename prefix is *mytrace*.
The AIX trace buffer size in Kilobytes. This value can be at most 1024, which is the default.	**bufsize**	`pct> trace set bufsize 1000`

The type of events (MPI events, process dispatch events, and CPU idle events) that are traced. By default, MPI and process dispatch events are traced. Tracing process dispatch events and CPU idle events can result in larger trace files, but the additional information can provide useful context for the MPI information collected.	**event**	`pct> trace set event mpi`
		`pct> trace set event process`
		`pct> trace set event idle`

To specify:	Use this clause of the trace set subcommand:	For example:
The maximum trace file size in Megabytes. This can be any value between 2 and 2048 inclusive. The default is 20.	**logsize**	`pct> trace set logsize 25`

Adding MPI Trace Probes to Processes: By adding MPI trace probes to processes, you can trace such MPI events as collective communication, point-to-point communication, and one-sided communication.

To add MPI trace probes, you'll need to know the specific MPI probe type identifier or name as returned by the **trace show** subcommand. To list the available MPI probe type identifiers and names, specify the **probetypes** clause on the **trace show** subcommand:

```
pct> trace show probetypes
MPI Id MPI Name    Description
------ ----------- -------------------------------------
0      all         all MPI events
1      blkcollcomm blocking collective communication
2      pttopt      point-to-point communication
3      onesided    one-sided communication
4      commgroup   communication groups
5      topo        topologies
6      collcomm    non-blocking collective communications
7      env         environmental
8      data        data type
9      file        file
10     info        information
11     comm        communicators
12     wait        wait calls
13     test        test calls
pct>
```

Once you have the probe type information, you can use the **trace add** subcommand to add one or more probe types to one or more processes. You can add the probes at the file level, in which case the MPI events for the entire file will be traced, or at the function level. If that granularity is not small enough, and you want to trace only a portion of a function, you can use special markers to force tracing on and off at particular points.

By default, the **trace add** subcommand acts upon the current task group. Unless you have specified another task group to be the current task group (as described in "Grouping Tasks of a POE Application" on page 108), the current task group will be the task group *connected*. The task group *connected* is created automatically by the PCT when you either connect to or load an application (as described in "Connecting to a Running Application" on page 111 and "Loading and Starting a New Application" on page 110). The task group *connected* consists of all connected tasks in a POE application. If you are instrumenting a serial application, you do not need to concern yourself with task groups. If you are instrumenting a POE application, however, it is useful to understand the concept of task groups as described in "Grouping Tasks of a POE Application" on page 108.

Note: The set of tasks in which you will add the probes cannot include different executables in an MPMD application. For example, if an MPMD application

consists of executables *a.out* and *b.out*, then this command cannot be applied to a task group that contains both *a.out* and *b.out* tasks.

If you are tracing at the file level, you'll need to specify the files using either the **file** or **fileid** clause on the **trace add** subcommand. To do this, you'll need the file identifier or file name information as returned by the **file** subcommand. To list all available source files in the current task group:

```
pct> file "*"
Tid File Id File Name Path
--- ------- --------- -------------
0   0       bar.c     ../../lib/src
0   1       foo1.c    ../../lib/src
0   2       foo2.c    ../src
pct>
```

To add a certain type of MPI probe, you supply the **trace add** subcommand with the MPI probe type and file information. You can specify the MPI probe type by supplying the:

- MPI probe type identifier using the **mpiid** clause
- MPI probe type name using the **mpiname** clause

Similarly, you can specify the file information by supplying the:

- file identifier using the **fileid** clause
- file name using the **file** clause and a regular expression

For example:

```
pct> trace add mpiid 0 to fileid 0

pct> trace add mpiname all to file "bar.c"
```

You can also specify multiple MPI probe types or multiple files:

```
pct> trace add mpiid 1,2 to fileid 0,1

pct> trace add mpiname collcom,pttopt to file "bar.c","f*"
```

If you would like to trace at a function level rather than tracing an entire file, you need to specify the function(s) using either the **function** or **funcid** clause. You'll need the function identifier or function name information as returned by the **function** subcommand. To list all functions in the file *bar.c*:

```
pct> function file "bar.c" "*"
Tid File Id Function Id File Name Function Name
--- ------- ----------- --------- -------------
0   1       0           bar.c     func0
0   1       1           bar.c     func1
pct>
```

Note:

If you wish to instrument a particular function, but do not know which file the function is located in, you can use the **find** subcommand. For example, to search all files in task 0 for functions that match the regular expression *comp**:

```
pct> find task 0 function "comp*"
Tid File Id File Name Function Name
--- ------- --------- -------------
```

```
0   23      main.c    compute
0   23      main.c    compare
0   25      sort.c    compare2
pct>
```

You can then specify the function on the **trace add** subcommand:

```
pct> trace add mpiid 0 to file "bar.c" function "func0"
```

You can also specify multiple functions:

```
pct> trace add mpiid 0 to file "bar.c" function "*"

pct> trace add mpiid 0 to file "bar.c" function "func0","func1"
```

Removing MPI Trace Probes From Processes: When you issue the **trace add** subcommand to install MPI trace probes, the probes are given a unique probe identifier. You can use the probe identifier on the **trace remove** subcommand to remove the probes. To ascertain the probe identifier, use the **trace show** subcommand with its **probes** clause as in:

```
pct> trace show probes
Probe Id Command
-------- ------------------------------------------------------------------
0        trace add mpiid 0 to file "prod_cons.c" function "alarm_handler"
1        trace add mpiid 0 to file "prod_cons.c" function "consume"
pct>
```

To remove the probe set whose probe identifier is 0:

```
pct> trace remove probe 0
```

Adding User Markers to Processes: *User markers* are special types of probes that you can install at specific instrumentation points in your application code. You can:

- Mark events of interest (such as program function calls) using a *simple marker*. A simple marker will appear in the trace record as a single point; its position gives you a frame of reference when analyzing the trace record in a graphical visualization tool like Jumpshot.
- Mark a state of interest using a *begin state marker* and an *end state marker*. A state marked by begin and end state markers will appear in the trace record as a region. Like the simple markers, this gives you a frame of reference when analyzing the trace record in a graphical visualization tool like Jumpshot.
- Force tracing on or off using a *trace on marker* or a *trace off marker*.

To install a user marker, you'll need to identify not only the file and function, but also the instrumentation point at which you want the probe installed. To list instrumentation points, issue the **point** subcommand.

```
pct> point task 0 file "bar.c"
Tid File Id Function Id Point Id Point Type Callee Name
--- ------- ----------- -------- ---------- ------------
0   54      0           0        0
0   54      0           1        2          printf
0   54      0           2        3          printf
0   54      0           3        2          MPI_Abort
0   54      0           4        3          MPI_Abort
0   54      0           5        1
0   54      1           0        0
0   54      1           1        2          printf
0   54      1           2        3          printf
0   54      1           3        2          printf
0   54      1           4        3          printf
0   54      1           5        2          MPI_Recv
```

```
0   54      1           6        3          MPI_Recv
0   54      1           7        2          consume_data
0   54      1           8        3          consume_data
0   54      1           9        2          printf
0   54      1           10       3          printf
0   54      1           11       1
pct>
```

To understand the point type number returned by the **point** command, issue the **show points** command.

```
pct> show points
Point Type  Point Name
----------  ----------
0           function entry
1           function exit
2           before callsite
3           after callsite
pct>
```

To:	Use:	For example:
mark a state of interest.	the **simplemarker** clause on the **trace add** subcommand.	`pct> trace add simplemarker "simple" to file "bar.c" funcid 0 pointid 0`

mark a region	the **beginmarker** and **endmarker** clauses on the **trace add** subcommand. You must mark the beginning and end of the range with the same ″marker name″ (a string that will be used to identify the user state in the trace record). You can only use a particular name for one begin marker/end marker pair. The state will appear in the trace record as a region.
	
	You should place all markers after the target application's call to *MPI_init* (which initializes MPI), and before the call to *MPI_Finalize* (which terminates MPI processing). For more information in the *MPI_init* and *MPI_Finalize* calls, refer to the *IBM Parallel Environment for AIX: MPI Programming Guide* or the *IBM Parallel Environment for AIX: MPI Subroutine Reference*.
	
	When marking a region, you must ensure that the begin and end state markers are placed so that if either marker is reached during execution, the other marker will also be reached. If you nest region markers, you must also ensure that the regions are properly nested. In other words, the inner region should be fully enclosed by the outer region. If you do not follow these guidelines, and the begin and end state markers are not correctly nested, you will get an error when you run the **uteconvert** utility. For more information on the **uteconvert** utility, refer to "Creating, Converting, and Viewing Information Contained In, UTE Interval Files" on page 126.
		`pct> trace add beginmarker "green" to file "bar.c" funcid 1 pointid 0`
		
		`pct> trace add endmarker "green" to file "bar.c" funcid 1 pointid 1`

force tracing on or off	the **traceon** or **traceoff** clause on the **trace add** subcommand.	`pct> trace add traceoff to file "bar.c" funcid 0 pointid 0`
		
		`pct> trace add traceon to file "bar.c" funcid 0 pointid 1`

Removing User Markers From Processes: When you issue the **trace add** subcommand to install a custom user marker, the marker is given a unique marker identifier. You can use this marker identifier on the **trace remove** subcommand to

remove the markers. To ascertain the marker identifier, use the **trace show** subcommand with its **markers** clause as in:

```
pct> trace show markers
Marker Id Command
--------- -------
0         trace add simplemarker "simple" to file "bar.c" funcid 0 pointid 0
1         trace add beginmarker "green" to file "bar.c" funcid 1 pointid 0
2         trace add endmarker "green" to file "bar.c" funcid 1 pointid 1
pct>
```

To remove the marker whose identifier is 2:

```
> trace remove marker 2
```

Collecting Hardware and Operating System Profile Information

Using the PCT, you can collect hardware and operating system profiles for analysis within the PVT.

The profile information collected is stored in netCDF (network Common Data Form) format on each node running instrumented processes. The PVT can read netCDF files and summarize the profile information in reports. For more information on using the PVT to read netCDF files output by the PCT, refer to “Using the Profile Visualization Tool” on page 131.

Before you can use any of the profile collection subcommands described in this section, you must first specify that you are collecting hardware profile information rather than MPI and user event traces. Refer to “Selecting the Type of Probe Data To Be Collected” on page 115 for more information. Once you have indicated that you’ll be collecting hardware profile information, you can select the output location for the netCDF files generated by the PCT. To do this, you simply supply an output directory and ″base name″ (file prefix) for the netCDF files. Refer to “Setting the Output Location for the netCDF Files Generated” for more information.

Setting the Output Location for the netCDF Files Generated: The hardware profile information is saved as a separate netCDF file on each node running instrumented processes. Using the **profile set path** subcommand, you can specify the output location and ″base name″ file prefix for these files. For example:

```
pct> profile set path "profile/output"
```

Adding Hardware Profile Probes to Processes: By adding hardware profile probes to processes, you can collect hardware and operating system information such as elapsed wall-clock time, process resource usage, and hardware counters. To add hardware profile probes, you need to know the specific probe type identifier or name as returned by the **profile show** subcommand. To list available probe type identifiers and names, specify the **probetypes** clause on the **profile show** subcommand.

For example:

```
pct> profile show probetypes
Prof Id Prof Name Description
------- --------- ----------------
0       wclock    wall clock
1       rusage    resource usage
2       hwcount   hardware counter
pct>
```

For hardware counters, you can also display a list of the specific hardware counter information you can collect. The list of available hardware counter groups will differ depending on whether the current or supplied task group:

- has tasks running only on 604e CPUs
- has tasks running only on 630 CPUs

If the current or supplied task group has tasks running on mixed CPUs, then no hardware counters are available, and so none will be listed.

To list available hardware counter groups, specify the **probetype hwcount** clauses on the **profile show** subcommand:

```
pct> profile show probetype hwcount
Prof Type Name    Description
--------- ------- --------------------------------
0         FPU     FPU, FXU, and LSU operations
1         Branch  Branch operations
2         L1_TLB  L1 cache and TLB operations
3         L2      Prefetch and L2 cache operations
4         Fpop    Floating-point operations
5         xFPU    FPU, FXU, LSU, and BPU operations
pct>
```

The hardware counter groups you see listed are, by default, the hardware counter groups we have created. For more information on the counters included in each group, and for information on creating your own counter groups, refer to "Appendix E. Understanding and Creating PCT Hardware Counter Groups" on page 225.

Once you have the probe type and hardware counter information, you can use the **profile add** subcommand to add one or more probe types to one or more processes. You can add the probes at the file level, in which case profile information for the entire file will be produced, or at the function level.

By default, the **profile add** subcommand acts upon the current task group. Unless you have specified another task group to be the current task group (as described in "Grouping Tasks of a POE Application" on page 108), the current task group will be the task group *connected*. The task group *connected* is created automatically by the PCT when you either connect to or load an application (as described in "Connecting to a Running Application" on page 111 and "Loading and Starting a New Application" on page 110). The task group *connected* consists of all connected tasks in a POE application. If you are instrumenting a serial application, you do not need to concern yourself with task groups. If you are instrumenting a POE application, however, it is useful to understand the concept of task groups as described in "Grouping Tasks of a POE Application" on page 108.

Note: The set of tasks in which you will add the probes cannot include different executables in an MPMD application. For example, if an MPMD application consists of executables *a.out* and *b.out*, then this command cannot be applied to a task group that contains both *a.out* and *b.out* tasks.

If you are collecting profile information at the file level, you'll need to specify the files using either the **file** or **fileid** clause on the **profile add** subcommand. To do this, you'll need the file identifier or file name information as returned by the **file** subcommand. To list all available source files in the current task group:

```
pct> file "*"
Tid File Id File Name Path
--- ------- --------- -------------
0   0       bar.c     ../../lib/src
0   1       foo1.c    ../../lib/src
0   2       foo2.c    ../src
pct>
```

To add a certain type of profile probe, you can supply the **profile add** subcommand with the profile probe type and option information, as well as the file information. You can specify:

- the profile probe type by supplying the:
  - profile probe type identifier using the **profid** clause
  - profile probe type name using the **profname** clause
- the hardware profile group using the **groupid** or **groupname** clause
- the file information by suppling the:
  - file identifier using the **fileid** clause
  - file name using the **file** clause and a regular expression

For example:

```
pct> profile add profname wclock to fileid 0

pct> profile add profid 0 to file "bar.c"

pct> profile add profname hwcount groupid 2 to fileid 3
```

You can also specify multiple profile probe types or multiple files:

```
pct> profile add profname wclock profname hwcount groupid 2 to fileid 3,4
```

If you would like to collect profile information at the function level (instead of collecting profile information for an entire file), you'll need to specify the function(s) using either the **function** or **funcid** clause. You'll need the function identifier or function name information as returned by the **function** subcommand. To list all the functions in the file *bar.c*:

```
pct> function file "bar.c" "*"
Tid File Id Function Id File Name Function Name
--- ------- ----------- --------- -------------
0   1       0           bar.c     func0
0   1       1           bar.c     func1
pct>
```

You can specify the function on the **profile add** subcommand using its identifier or name:

```
pct> profile add profname wclock to file "bar.c" function "func0"

pct> profile add profname wclock to file "bar.c" funcid 0
```

You can also specify multiple functions:

```
pct> profile add profname wclock to file "bar.c" funcid 0,1

pct> profile add profname wclock to file "bar.c" function "*"

pct> profile add profname wclock to file "bar.c" function "func0","func1"
```

Removing Hardware Profile Probes From Processes: When you issue the **profile add** subcommand to install profile probes, the probes are given a unique probe identifier. You can use this probe identifier on the **profile remove** subcommand to remove the probes. To ascertain the probe identifier, use the **profile show** subcommand with its **probes** clause as in:

```
pct> profile show probes
Probe Id Command
-------- ------------------------------------------------------------------
0        profile add profid 0 to file "prod_cons.c" function "alarm_handler"
1        profile add profid 0 to file "prod_cons.c" function "consume"
pct>
```

To remove the probe set whose identifier is 0:

```
pct> profile remove probe 0
```

Terminating Connected Processes

The PCT enables you to terminate execution of connected processes by issuing the **destroy** subcommand. You might, for example, wish to terminate execution of your target application after you have finished examining it. By default, the **destroy** subcommand acts upon the current task group. Unless you have specified another task group to be the current task group (as described in “Grouping Tasks of a POE Application” on page 108), the current task group will be the task group *connected*. The task group *connected* is created automatically by the PCT when you either connect to or load an application (as described in “Connecting to a Running Application” on page 111 and “Loading and Starting a New Application” on page 110). The task group *connected* consists of all connected tasks in a POE application. If you are instrumenting a serial application, you do not need to concern yourself with task groups. If you are instrumenting a POE application, however, it is useful to understand the concept of task groups as described in “Grouping Tasks of a POE Application” on page 108.

Note: When working with a POE application, be aware that terminating any process of the application will cause POE to terminate **all** of the application’s processes. This termination of all processes is a function of POE, not of the PCT. For more information, refer to the manual *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

To terminate execution of all tasks in the current task group:

```
pct> destroy
```

To terminate execution of tasks in a specific task group (in this case, the task group *connected*), use the **group** clause on the **destroy** subcommand.

```
pct> destroy group connected
```

To terminate a specific set of tasks in a POE application, use the **task** clause on the **destroy** subcommand. To determine how many tasks are available, you can use the **show group** subcommand to list the tasks in the task group *all*:

```
pct> show group all
Tid Program Name             Host              Cpu Type State
--- ------------------------ ----------------- -------- ------
0   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
1   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
2   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
3   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
.
.
.
pct> destroy task 1,3
```

You can also, optionally, terminate execution of all connected tasks when exiting the PCT. To do this, use the **exit** command with its **destroy** clause (as described in “Exiting the Performance Collection Tool” on page 125).

Disconnecting From the Application

Once you are through examining a particular application, or particular tasks in an application, you can disconnect from the application or application tasks by issuing the **disconnect** subcommand. Once a process is disconnected, the PCT will no longer be able to control execution of, or instrument, the process unless it

reconnects to the process. By default, the **disconnect** subcommand acts upon the current task group. Unless you have specified another task group to be the current task group (as described in “Grouping Tasks of a POE Application” on page 108), the current task group will be the task group *connected*. The task group *connected* is created automatically by the PCT when you either connect to or load an application (as described in “Connecting to a Running Application” on page 111 and “Loading and Starting a New Application” on page 110). The task group *connected* consists of all connected tasks in a POE application. If you are instrumenting a serial application, you do not need to concern yourself with task groups. If you are instrumenting a POE application, however, it is useful to understand the concept of task groups as described in “Grouping Tasks of a POE Application” on page 108.

To disconnect all tasks in the current task group:

```
pct> disconnect
```

To disconnect tasks in a specific task group (in this case, the task group *connected*), use the **group** clause on the **disconnect** subcommand.

```
pct> disconnect group connected
```

To disconnect a specific set of tasks in a POE application, use the **task** clause on the **disconnect** subcommand. To determine how many tasks are available, you can use the **show group** subcommand to list the tasks in the task group *all*:

```
pct> show group all
Tid Program Name             Host              Cpu Type State
--- ---------------------- ---------------- -------- ------
0   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
1   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
2   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded
3   /home/strofino/prod_cons pe04.pok.ibm.com Unknown  Loaded.
.
.
pct> disconnect task 1,3
```

Exiting the Performance Collection Tool

To exit the PCT and return to your AIX command prompt, issue the **exit** subcommand:

```
pct> exit
```

If you loaded the target application, it will be terminated when PCT exits. If you merely connected to the target application, you must explicitly instruct the PCT to terminate processes. To terminate execution of all connected processes as you exit the PCT, include the **destroy** clause on the **exit** subcommand.

```
pct> exit destroy
```

Creating and Running PCT Script Files

Using the command-line interface of the PCT, you are able to run a series of commands that are stored in a file. This file, called a ″PCT script file″ is a simple text file that lists a sequence of PCT commands that you want to run. Because PCT script files are reusable, they are ideal for situations where you have a set of commands you want to run during multiple PCT sessions. For example, you might want to create a PCT script file that loads and prepares an application so that you can then perform a variety of tasks on the prepared application.

To create a PCT script file, use any ASCII text editor. In the file, place one PCT command per line. You can add comment lines to the file using the # (pound sign) character. For example, here is a simple PCT script file.

```
# This example uses the 'chaotic' application from the DPCL samples.
# The script loads a four-way chaotic application, inserts probes,
# starts the application, and then waits for the application to complete
load poe exec /home/user/chaotic poeargs "-procs 4"
select trace
trace set path "/scratch/trace_out"
trace add mpiid 0 to file "chaotic.f"
start
wait
```

In the sample PCT script file shown above, note the use of the **wait** subcommand. You need to use the **wait** subcommand in PCT script files to prevent the PCT from exiting before it has collected probe data. The **wait** subcommand blocks the PCT's execution so that it can wait for asynchronous events (such as a task terminating) to occur. When one of these asynchronous events occurs, the PCT resumes execution and returns the event that occurred. Be aware that the **wait** subcommand is intended for use only within PCT script files; it is not intended for interactive command-line sessions.

To run the script file, you can either use the **-s** option of the **pct** command when starting the tool (as described in "Starting the Performance Collection Tool In Command-Line Mode" on page 107), or you can use the **run** subcommand of the **pct** command. For example, to run the PCT script file *myscript.cmd* when starting the tool, you would enter the following at the AIX command prompt:

```
pct -c -s myscript.cmd
```

Alternatively, you could run the *myscript.cmd* script file using the **run** subcommand. For example:

```
pct> run "myscript.cmd"
```

Creating, Converting, and Viewing Information Contained In, UTE Interval Files

When you collect MPI and user event traces using the PCT (as described in "Using the Performance Collection Tool" on page 100), the collected information is saved, on each machine running instrumented processes, as a standard AIX event trace file. In order to view the information contained in these standard AIX trace files, you will first need to convert them into UTE (Unified Trace Environment) interval files. While an AIX event trace file has a time stamp indicating the point in time when an event occurred, UTE interval files take this information to also determine how long an event lasts. Because they include this duration information, UTE interval files are easier to visualize than traditional AIX event trace files. The UTE utilities are:

- The **uteconvert** utility which coverts AIX event trace files into UTE interval trace files.
- The **utemerge** utility which merges multiple UTE interval files into a single UTE interval file.
- The **utestats** utility which generates statistics tables from UTE interval files.
- The **slogmerge** utility which converts and merges UTE interval files into a single SLOG file for analysis within Argonne National Laboratory's Jumpshot tool.

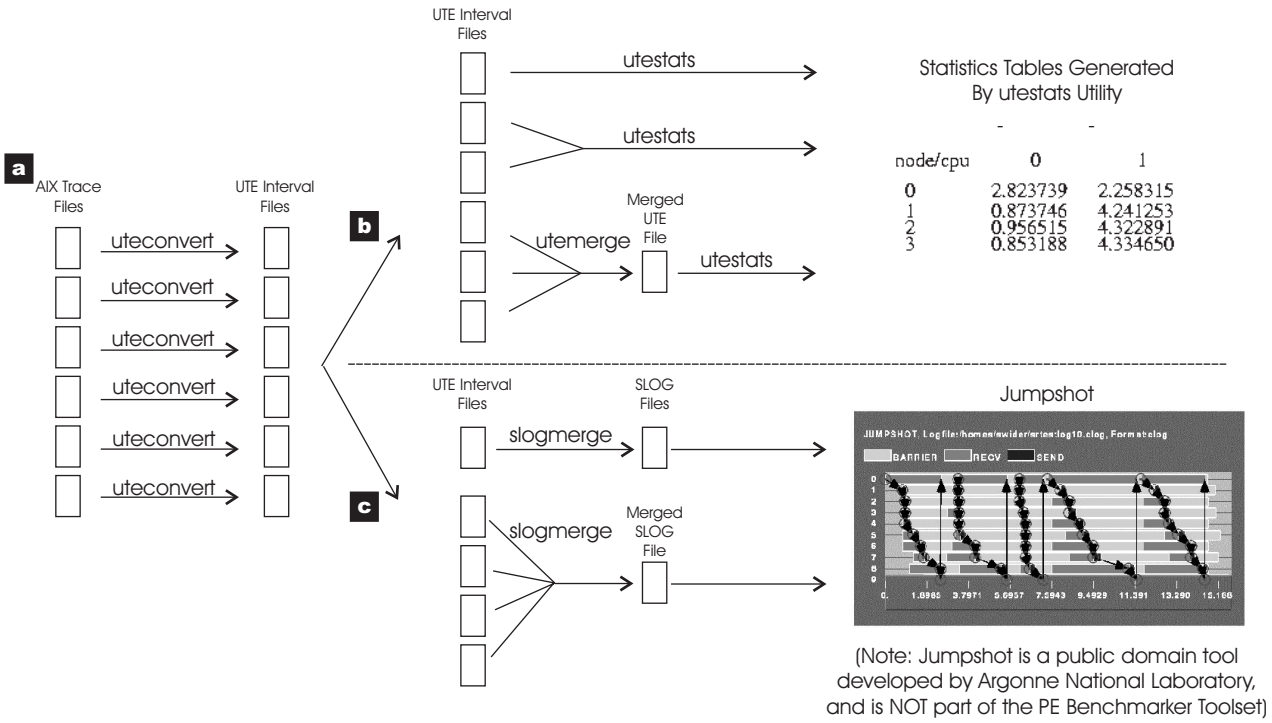


Figure 37. Unified Trace Environment (UTE) Utilities

The preceding figure illustrates the UTE utilities you can use to either generate statistics tables from UTE interval files or view statistics graphically using Argonne National Laboratory's Jumpshot tool. Regardless of whether you want to view the statistics in simple tables or graphically in Jumpshot, the first thing you'll need to do is use the **uteconvert** utility to create UTE interval files from the AIX trace files (**a**). (See "Converting AIX Trace Files Into UTE Interval Trace Files" on page 128 for more information.) Then, if you want to view the statistics in simple tables (**b**), you can use the **utestats** utility. You can optionally merge multiple UTE files into a single UTE file using the **utemerge** utility before using the **utestats** utility to generate the statistics tables. (See "Generating Statistics Tables From UTE Interval Trace Files" on page 128 for more information.) If you instead want to view the information contained in the UTE interval files graphically (**c**), you can convert them into SLOG files using the **slogmerge** utility. The SLOG files are readable by Argonne National Laboratory's Jumpshot Tool. (See "Converting UTE Interval Files Into SLOG Files Required By Argonne National Laboratory's Jumpshot Tool" on page 130 for more information.)

Note: The UTE utilities are intended only for the AIX event trace files generated when you collect MPI and user event traces with the PCT. If you instead collect hardware and operating system profiles, the information is output by the PCT as netCDF (network Common Data Form) files and these UTE utilities are not necessary. Instead, the netCDF files can be read directly into the PVT as described in "Using the Profile Visualization Tool" on page 131.

The following sections provide an overview of the UTE utilities. Note, however, that this section does not attempt to describe all the options available when using these utilities. For complete reference information on any of the utilities described in this section, refer to their man pages contained in "Appendix A. Parallel environment tools commands" on page 139.

Converting AIX Trace Files Into UTE Interval Trace Files

Regardless of whether you want to view the statistics you have collected in simple tables, or graphically in Jumpshot, the first thing you'll need to do is use the **uteconvert** utility to create UTE interval files from the AIX trace files generated by the PCT. When you collect MPI and user event traces, the collected information is saved, on each machine running instrumented processes, as a standard AIX event trace file. The names of these individual trace files will consist of a common ″base name″ that you specified using the PCT, followed by a node-specific suffix supplied by the tool itself. Using the **uteconvert** utility, you can convert either a single AIX trace file into a UTE interval file, or a set of AIX trace files with the same prefix into a set of UTE interval files.

To convert a single AIX trace file into a UTE interval file, simply pass the **uteconvert** utility the name of the trace file located in the current directory. For example, to convert the AIX trace file *mytrace* into a UTE interval trace file, enter:

```
uteconvert mytrace
```

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

```
uteconvert -o outute mytrace
```

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

```
uteconvert -n 5 mytraces
```

You can optionally use the **–o** option to specify a file name prefix for the resulting UTE interval files.

```
uteconvert -n 5 -o outute mytraces
```

When you use the **-n** option, make sure you do not have any old AIX trace files from previous executions of the program still in the directory. The **uteconvert** utility will process the first *n* trace files it finds that match the base name prefix.

For complete reference information on the **uteconvert** utility, refer to its man page in "Appendix A. Parallel environment tools commands" on page 139. If you want to view the statistics information contained in the UTE file(s) in simple tables, refer to "Generating Statistics Tables From UTE Interval Trace Files". If you want to view the statistics information contained in the UTE file(s) graphically, refer to "Converting UTE Interval Files Into SLOG Files Required By Argonne National Laboratory's Jumpshot Tool" on page 130.

Generating Statistics Tables From UTE Interval Trace Files

Once you have created UTE interval trace files (as described in "Converting AIX Trace Files Into UTE Interval Trace Files"), you can generate statistical tables from them using the **utestats** utility. In addition to giving you a simple alternative to graphical analysis, the **utestats** utility can help you identify which traces you want to view in a graphical visualization tool like Jumpshot. This is useful, because you are often unable to view all process threads in a graphical visualization tool. Jumpshot, for example, supports only 64 threads. Using the **utestats** utility, you can determine which threads are of interest. In addition, if you do not wish to use a graphical visualization tool, you can analyze traces extensively using the **utestats** utility alone.

By default, six two-dimensional tables are generated. These tables are:

- Time Bin vs. Node
- Thread vs. Event Type
- Event Type vs. Thread
- Node vs. Event Type
- Event Type vs. Node
- Node vs. Processor

The computed statistic for all tables is the sum or the duration. A Node vs. Processor table would look like the following (where tabs have been replaced by spaces to make the column alignment clearer). The unit of measurement is seconds, so, for example, the accumulated duration of all interval records for CPU 1 of node 0 was 2.258315 seconds.

```
node/cpu            0               1
   0             2.823739        2.258315
   1             0.873746        4.241253
   2             0.956515        4.322891
   3             0.853188        4.334650
```

You can generate these statistics tables for a single UTE interval file or multiple UTE interval files. You can also generate these statistics tables for a merged UTE interval file. A merged UTE interval file is one that consists of multiple UTE interval files that have been merged into one file by the **utemerge** utility.

For example, to generate the statistics tables for the UTE interval file *mytrace.ute*, you would enter:

```
utestats mytrace.ute
```

By default, the statistics tables will be printed to standard output. You can, however, redirect them to a file using the **-o** option on the **utestats** command. For example, to redirect the statistics tables output by the **utestats** utility to the file *stattables*, you would enter:

```
utestats -o stattables mytrace.ute
```

As already stated, you can also specify multiple UTE interval files from which the statistics should be generated.

```
utestats mytrace.ute mytrace2.ute mytrace3.ute
```

Rather than specify multiple UTE interval trace file names on the **utestats** command, you could instead use the **utemerge** utility to first merge the multiple UTE interval trace files into a single UTE interval trace file. To do this, you use the **-n** option on the **utemerge** command to indicate the number of files you want to merge, and supply the common ″base name″ prefix shared by the files. For example:

```
utemerge -n 3 mytrace
```

The merged UTE interval file generated by the **utemerge** utility will, by default, be named *trcfile.ute*. To specify your own output file name, use the **-o** option.

```
utemerge -n 3 -o mergedtrc.ute mytrace
```

When you use the **-n** option, make sure you do not have any old UTE interval files from previous executions of the program still in the directory. The **utemerge** utility will process the first *n* interval files it finds that match the base name prefix.

You can then generate statistics for the merged UTE interval file using the **utestats** command.

```
utestats mergedtrc.ute
```

For complete reference information on the **utestats** and **utemerge** utilities, refer to their man pages in “Appendix A. Parallel environment tools commands” on page 139.

Note: Argonne National Laboratory’s Jumpshot Tool also includes a statistics view feature that displays the same information as the **utestats** command generates. Jumpshot also has the ability to display statistics information graphically. The Jumpshot Tool is described next in “Converting UTE Interval Files Into SLOG Files Required By Argonne National Laboratory’s Jumpshot Tool”.

Converting UTE Interval Files Into SLOG Files Required By Argonne National Laboratory’s Jumpshot Tool

If you would like to view the traces collected by the PCT graphically, you can use the Jumpshot tool developed by Argonne National Laboratory. While Jumpshot is a public domain tool and **not** part of the PE Benchmarker Toolset, we do provide a utility — **slogmerge** — for converting UTE interval files into the SLOG files required by Jumpshot. You can use the **slogmerge** utility to:

- convert a single UTE interval file into a single SLOG file.
- merge multiple UTE interval files into a single SLOG file.

If you are dealing with a massively parallel job, it is unlikely that you will be able to display all the process threads in Jumpshot. In fact, Jumpshot supports only 64 threads. Rather than merge all the trace files generated from such a job, you will instead want to merge selected trace files. To determine which files to merge, you can first use the **utestats** utility (as described in “Generating Statistics Tables From UTE Interval Trace Files” on page 128) to determine the characteristics of the files. By analyzing the files first using the **utestats** utility, you can determine which files contain the interesting information that you want to merge and view in Jumpshot.

To convert a single UTE interval file into a single SLOG file, pass the **slogmerge** command the name of the file located in the current directory. For example:

```
slogmerge mytrace.ute
```

By default, the SLOG file output by the **slogmerge** utility will be *trcfile.slog*. Using the **-o** option on the **slogmerge** command, however, you can specify an output file name. For example:

```
slogmerge -o mergedtrc.slog mytrace.ute
```

To merge multiple UTE interval files into a single SLOG file, use the **-n** option to indicate the number of files to merge and pass the **slogmerge** utility the common ″base name″ prefix of the files. For example, to merge 3 files whose prefix is *mytrace*, enter:

```
slogmerge -n 3 mergedtrc.slog mytrace
```

When you use the **-n** option, make sure you do not have any old UTE interval files from previous executions of the program still in the directory. The **slogmerge** utility will process the first *n* interval files it finds that match the base name prefix.

For complete reference information on the **slogmerge** utility, refer to its man page in “Appendix A. Parallel environment tools commands” on page 139.

Using the Profile Visualization Tool

The PVT is a post-mortem analysis tool. It is designed to process profile data files generated by the PCT used in application profiling. For more information on the PCT, refer to “Using the Performance Collection Tool” on page 100. After processing profile data, you can view the results in the PVT’s graphical user interface display. You can also generate report and summary files. The PVT provides a command-line interface to process individual profile files directly into a summary file without initializing the graphic display. The command-line interface also enables you to generate textual profile reports. This section begins with a discussion of the PVT’s graphical user interface, followed by a description of the command-line interface.

Using the Profile Visualization Tool’s Graphical User Interface

The PVT provides a graphical user interface that enables you to process profile data files and view the results. The options available in the graphical user interface correspond to the commands available in the PVT’s command-line interface. For more information on the command-line interface, refer to “Using the Profile Visualization Tool’s Command Line Interface” on page 136.

Profile Visualization Tool (Graphical User Interface) Overview

The PVT’s graphical user interface allows you to process and view profile data. You can load one or more files for processing and view the results in a variety of ways. After initializing the graphical user interface, you can choose the appropriate options:

If:	Then:
You wish to load files for processing.	Select **File → Load...**. Doing this opens the Load Files panel. The Load Files panel will enable you to specify what files to load into the tool for processing. You can specify one or more individual profile files, or a summary profile file.
You wish to control the way profile data is presented.	Select the **View** option. Doing this opens the View menu. The View menu will enable you to specify how profile data is presented in the Main Display window. You can specify how to sort data, as well as show function call count and resource usage.
You wish to view selected objects.	Select the **Object** option. Doing this opens the Object menu. The Object menu will enable you to view information such as source code, profile data, and statistics reports for selected objects.
You wish to search for a text string.	Select **File → Find...** Doing this opens the Find panel. The Find panel will enable you to specify the text string for which you want to search.

You wish to generate reports of profile data.	Select the **Report** option. Doing this opens the Report menu. The Report menu will enable you to select and view a variety of reports, including function call count, CPU usage, and memory usage.
You wish to save summary data to a file.	Select **File → Save Statistic Summary...** Doing this opens the Save Statistic Summary panel. This panel will enable you to accept a user-specified file name. The statistic summary data of the input profile file or files will be written to the file.
You wish to export profile data to a file.	Select **File → Export...** Doing this opens the Export panel. This panel will enable you to accept a user-specified file name. The profile data that is currently loaded will be written to the file.
You wish to set user preferences.	Select **File → Preferences...** Doing this opens the Preferences panel. At this time, this panel will enable you to access only one option: source code search paths. There is a text field available that allows you to specify where the source code files reside.
You wish to exit the PVT.	Select **File → Exit...** Doing this closes the Main Display window and exits the PVT.

The following sections describe the graphical user interface in greater detail.

Starting the Profile Visualization Tool

You can start the PVT in either graphical-user-interface (GUI) mode or command-line mode. For instructions on starting the PVT in command-line mode, refer to “Using the Profile Visualization Tool’s Command Line Interface” on page 136. To start the PVT in graphical-user-interface mode:

Enter the pvt command at the AIX command prompt.

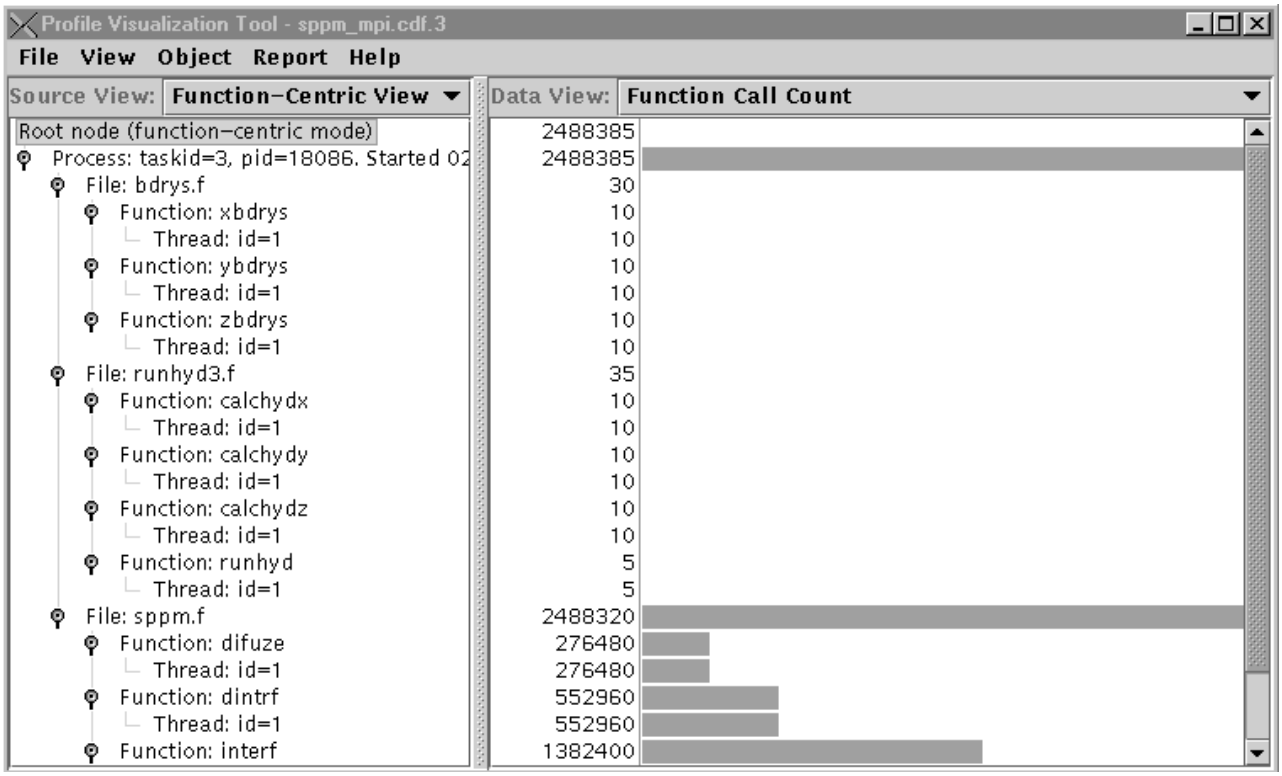
```
$ pvt
```

Doing this starts the PVT in graphical-user-interface mode and opens its first window – the Main Display.

To start the PVT in graphical-user-interface mode with input profile data loaded and showing in the Main Display window, enter:

```
$ pvt one_or_more_file_names
```

The following figure shows an example of the Main Display window with input profile data loaded.



The Main Display window shows a hierarchical list of all the functions being profiled. The window is divided into two panes, the left one for viewing source code structure and the right one for viewing profile data. Each pane has a corresponding menu: the **Source View** menu and the **Data View** menu. Both the Source View and Data View menus are grayed out if no input file is loaded. The two panes share the same vertical scroll bar and are scrolled together. You can resize the panes horizontally to change their relative proportion in the Main Display window.

The source code structure pane uses ASCII text to show the identifier of each displayed object. The profile data pane represents a selected profile data field, which uses a bar chart to show the profile data associated with each object. The data value is displayed in front of the bar. When you select an object in the source code structure pane, an object menu opens that provides some actions associated with the selected object. You left-click to select an object, and right-click to bring up the selected object's object menu. When you select an object, the **Object** menu in the Main Display window will become available also, providing the same functions as the popup object menu.

If you load a summary profile file to start the GUI, process objects are labeled as **summary process object** in order to distinguish them from the process objects available in an individual profile file. Each function object has a set of statistics records associated with each profile data field.

Following are explanations of the Source View and Data View menus.

Viewing Source Code Structure: The Source View is a drop-down menu with two options: a **Thread-Centric View** and a **Function-Centric View**. The same options are available under the **View** drop-down menu in the Main Display window. See "Viewing program variables" on page 24 for more information. If the input file

you are loading to start the GUI is a summary file, there will be no thread information in the file. The structure displayed will be the same no matter which view is used.

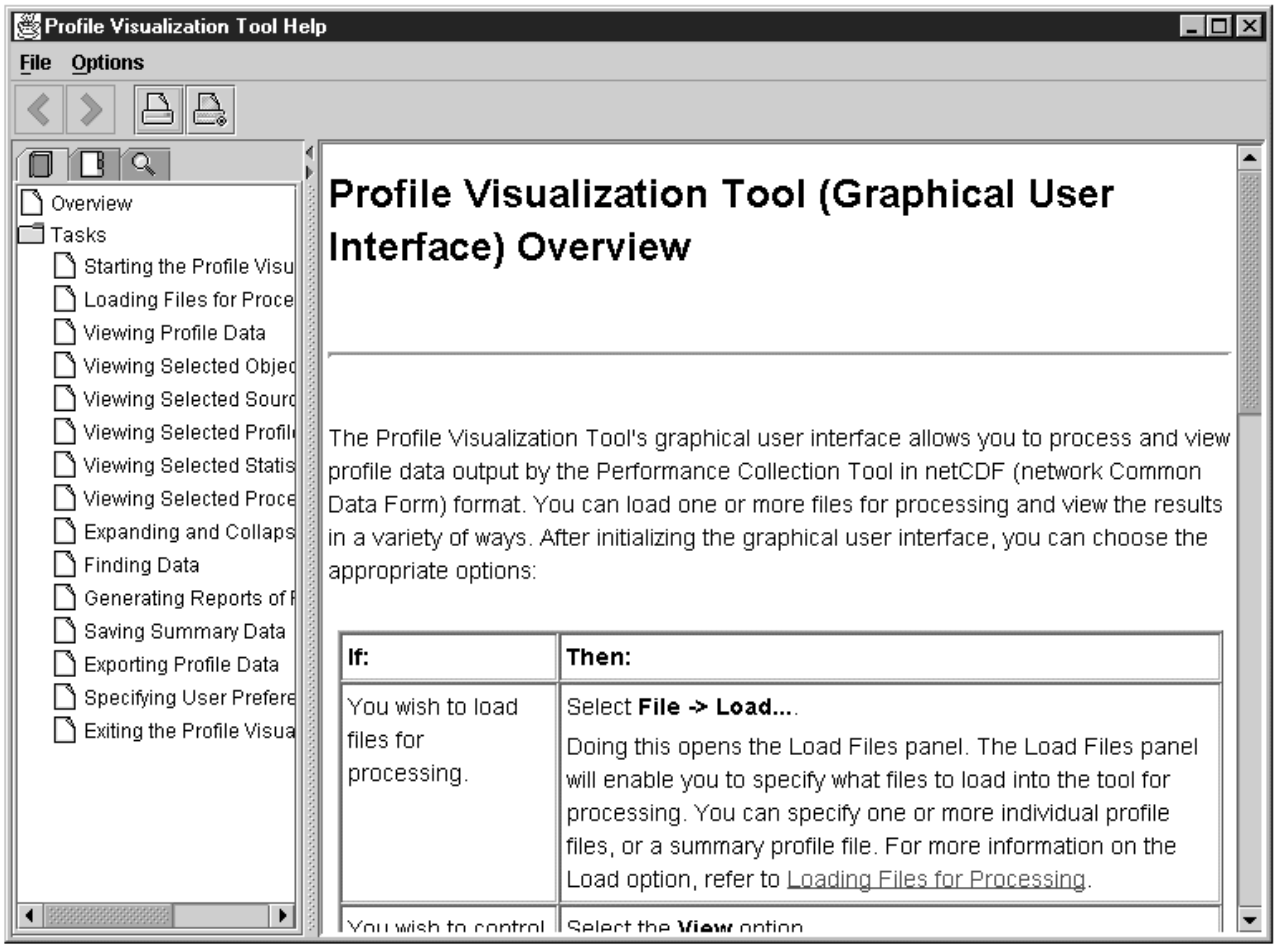
Viewing Selected Profile Data: The Data View is a drop-down menu that enables you to change the type of data to be shown in the Main Display window. The Data View menu options include the following categories:

- **Function Call Count**
- **Wall Clock Time**
- **Resource Usage**
- **Hardware Counters**.

You will find similar options available in the **View** drop-down menu. When a particular data type is unavailable in any of the input data files, its corresponding menu option in the View menu is grayed out. The Data View drop-down menu only shows the options that have corresponding values in the input data files. When a set of files is loaded, **Function Call Count** is the default field in the Data View menu.

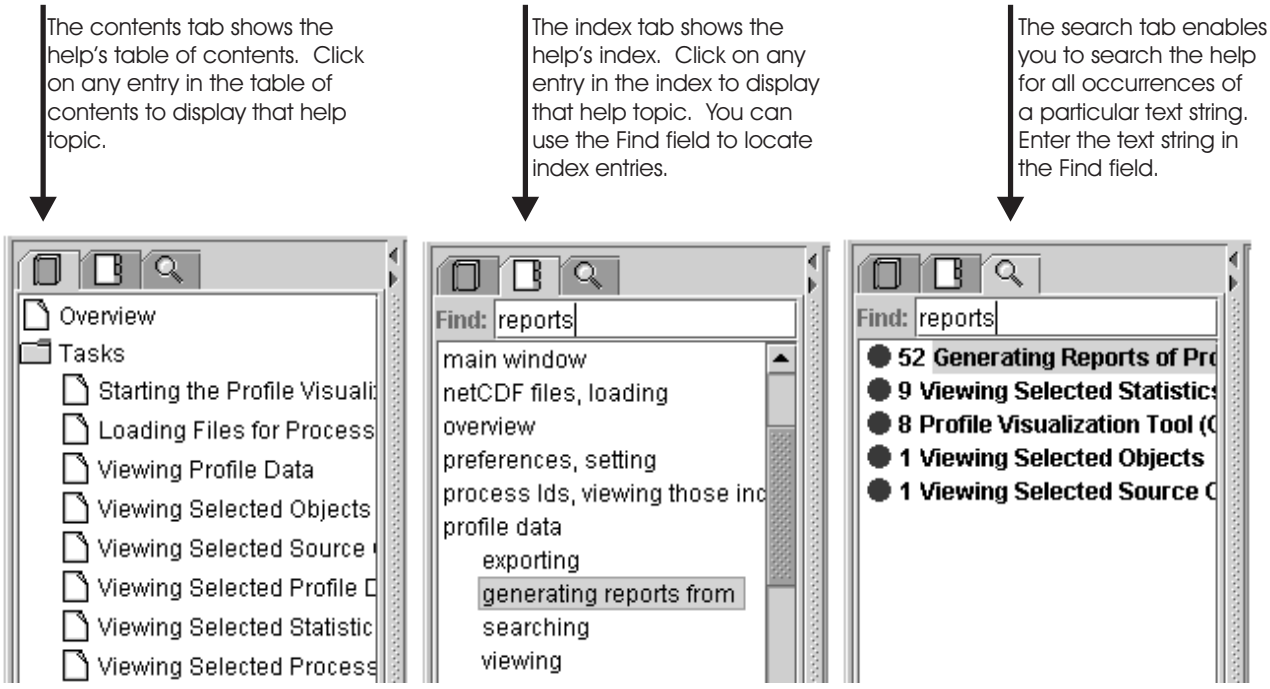
Accessing the Profile Visualization Tool's online help system

The PVT's graphical user interface has been designed to be intuitive and easy to use. However, if you do have any trouble, you can refer to the PVT's online help system. To access the tool's online help, select **Help → Contents** off the main window's menu bar. Many dialogs of the tool also provide **Help** buttons or menu items for starting the help system.



If you open the help from one of the PVT's dialogs, a help topic describing that dialog is displayed. If you open the help from the main window, a task overview topic is displayed.

The PVT help contains topics for each of the major tasks you can perform with the PVT. The left hand pane of the window enables you to navigate the help system to display the needed help topic in the right hand pane. There are three ways to navigate the help system — using the contents tab, using the index tab, or using the search tab:



- the contents tab is displayed by default. Simply click on any entry in the contents tab to display the help topic.
- the index tab shows an index of the entire help system. Simply click on any entry in the index to display its associated help topic. To search the index, type a string in the **Find** field and press **<enter>**. The first index entry containing the string is highlighted. Press **<enter>** again to search for the next occurrence of the string in the index.
- the search tab enables you to search the help for all occurrences of a text string. Simply type the string in the **Find** field and press **<enter>**. A list of all help topics containing the string is displayed. The topics are listed in descending order according to the number of occurrences of the string. The help topic with the most occurrences of the string is displayed by default.

Using the Profile Visualization Tool's Command Line Interface

The PVT provides a command-line interface that enables you to process profile files directly without initializing the graphical user interface. The subcommands available in the command-line interface correspond to the options available in the graphical user interface. For more information on the graphical user interface, refer to "Using the Profile Visualization Tool's Graphical User Interface" on page 131.

Profile Visualization Tool (Command Line Interface) Overview

The PVT's command-line interface allows you to process profile data directly without using the graphical user interface. After initializing the command-line interface, you can enter the appropriate subcommands that enable you to:

- Load files for processing
- Create a summary file of all the loaded data
- Generate textual reports of profile data
- Export profile data to a file.

The following sections describe the command-line interface in greater detail.

Starting the Profile Visualization Tool in Command-Line Mode

To start the PVT in command-line mode, enter:

```
pvt -c
```

Doing this starts a command-line session without associated profile data. To start a command-line session with associated profile data, enter:

```
pvt -c one_or_more_file_names
```

Once you start a command-line session, the command line prompt changes to **pvt>** and remains this way until you enter the **exit** command to end the command-line session.

The following sections describe the command-line mode subcommands.

Loading Files

You can load a set of profile data files into the session with the **load** command. Enter:

```
load one_or_more_file_names
```

If a set of data already exists, then the existing data is discarded and the newly loaded data becomes the current data to be used in future actions.

Creating a Summary File

You can create a summary file of all the loaded data with the **sum** command. Enter:

```
sum summary_file_name
```

The merged summary data is written to the file that you specify in the command, with a suffix of *.cdf* being appended to the specified file name.

Generating Reports

You can generate textual reports of profile data using the **report** command. You can specify several different options with the report command, depending on what type of output you want. To show a list of available report types, enter:

```
report list
```

The result will look something like:

- **[0] call_count:** function call count report
- **[1] wclock:** wall clock timer report
- **[2] ru_cpu:** CPU usage reports
- **[3] ru_mem:** memory usage report
- **[4] ru_paging:** paging activities reports
- **[5] ru_cswitch:** context switch activities reports
- **[6] pmc_cycle:** instructions per cycle hardware counter reports
- **[7] pmc_fpu:** floating-point hardware counter reports
- **[8] pmc_fxu:** fixed-point hardware counter reports
- **[9] pmc_branch:** branch hardware counter reports
- **[10] pmc_lsu:** load and store hardware counter reports
- **[11] pmc_cache:** cache hardware counter reports
- **[12] pmc_misc:** miscellaneous hardware counter reports

To generate all the available reports to a file, enter:

```
report output_file_name
```

To generate reports by report name to a file, enter:

```
report "one_or_more_report_names" output_file_name
```

For example:

```
report "wclock,ru_cpu" output
```

To generate reports by report id to a file, enter:

```
report "one_or_more_report_ids" output_file_name
```

For example:

```
report "1,2" output
```

The report names or report ids in double quotes must be separated by a comma, with no blank space in between. No matter how many reports are selected in one report command, all the reports are output to a single file specified in the report command.

Exporting Files

You can export profile data to a specified file using the **export** command. Enter:

```
export output_file_name
```

A suffix *.txt* will be appended to the specified file name.

The currently loaded profile data is written to the user-specified file in plain text format, so the data can be loaded easily into a spreadsheet tool like Lotus® 1-2-3®. The data that is loaded into the tool can be grouped into the following types of records:

- Profile-session record associated with each process (that is, profile session)
- Individual function or thread records
- Function statistics records.

Exiting the Profile Visualization Tool

You can end a command-line session with the **exit** command. Enter:

```
exit
```

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 “Conventions and terminology used in this book” on page xiv.

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.

pct

NAME

pct – Invokes the Performance Collection Tool (PCT) in either its graphical-user-interface or command-line mode.

SYNOPSIS

pct [**-c** [**-s** *script_file*]]

The **pct** command starts the PCT in either its graphical-user-interface mode, or, if the **-c** flag is specified, its command-line mode.

FLAGS

-c Specifies that the PCT should be started in command-line mode. Refer to "Subcommands of the pct command" on page 141 for information on the subcommands you can issue once the PCT is running in this mode.

-s *script_file*
When running in command-line mode, instructs the PCT to read its commands from the script file specified. When running in graphical user interface mode, you cannot use this option.

DESCRIPTION

The PCT is a highly scalable performance monitoring tool built on dynamic instrumentation technology — the Dynamic Probe Class Library (DPCL). Using the PCT, you can collect:

- MPI and user event traces for eventual analysis by either:
  - Jumpshot (a public-domain tool developed at Argonne National Lab).
    or
  - the **utestats** utility provided as part of the PE Benchmarker Toolset.

  Since the MPI and user trace information will be output as standard AIX trace files, we have also supplied, as part of the PE Benchmarker tool set, several utilities for converting the AIX trace files created by the PCT into a format readable by Jumpshot and the **utestats** utility.
- Hardware and operating system profiles for playback within the Performance Visualization Tool (as invoked by the **pvt** command).

The PCT can be run in either its graphical-user-interface mode, or, if the **-c** flag is specified, its command-line mode. The PCT's graphical user interface is built on top of its command-line interface; in other words, your manipulations of the graphical user interface are translated by the tool into **pct** subcommands. These subcommands are issued, and the information returned is used to update the graphical user interface. The **pct** subcommands that result from your interface interactions are displayed in an information area of the PCT's Main Window.

When running in command-line mode, you can optionally have the PCT read its commands from a script file. You can specify the script file using the **-s** option when issuing the **pct** command, or you can use the **run** subcommand.

The **pct** command's subcommands (for controlling the PCT in command-line mode) are listed alphabetically under "Subcommands of the pct command" on page 141.

EXAMPLES

To start the PCT in graphical-user-interface mode:

```
pct
```

To start the PCT in command-line mode:

```
pct -c
```

To start the PCT in command-line mode, and read commands from the script file *myscript.cmd*.

```
pct -c -s myscript.cmd
```

RELATED INFORMATION

Commands: **uteconvert**(1), **pvt**(1), **slogmerge**(1), **utemerge**(1), **utestats**(1)

Subcommands of the pct command

comment subcommand (of the pct command)

[*comment-string*]

The **comment** subcommand is intended for use within script files you write, and is not intended for interactive command-line sessions. Essentially, the # (pound sign) character instructs the PCT to ignore the rest of the line.

comment-string
Is any comment you want to add to the file.

For example, the following PCT script file contains three comment lines to explain the purpose of the script:

```
# This example uses the 'chaotic' application from the DPCL samples.
# The script loads a four-way chaotic application, inserts probes,
# starts the application, and then waits for the application to complete
load poe exec /home/user/chaotic poeargs "-procs 4"
select trace
trace set path "/scratch/trace_out"
trace add mpiid 0 to file "chaotic.f"
start
wait
```

connect subcommand (of the pct command)

connect [{**pid** *process_id* | **poe pid** *poe_process_id*} | **task** *task_list* | **group** *task_group_name*]

The **connect** subcommand connects the PCT to an existing application. Using this subcommand, you can connect to a single application process, or the controlling, ″home node″ process in a POE application. Once you are connected to a controlling POE home node process, you can reissue this subcommand to connect to one or more of the POE application's tasks.

pid *process_id*
Specifies the process ID of a single application process to connect.

poe pid *poe_process_id*
Indicates that you are connecting a POE process, and specifies the process ID of the POE home node process (the executing instance of the **poe**

command). Only the controlling POE process is connected. To connect to one or more of the POE application's tasks, reissue the **connect** subcommand.

task *task_list*
Specifies a list of POE tasks to connect. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refer to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
Specifies the name of a task group. To connect to all tasks in a POE application, you can specify the task group *all*, which will have been created by the PCT when you connected to the controlling, home node, POE process. Refer to the **group** subcommand for information on creating task groups.

For example, to connect to the application process whose AIX process ID is 12345:

```
pct> connect pid 12345
```

To connect to the POE ″home node″ process whose AIX process ID is 12345:

```
pct> connect poe pid 12345
```

The preceding example connects to just the controlling, home node, process in a POE application. To now connect to all of the tasks in the POE application:

```
pct> connect group all
```

destroy subcommand (of the pct command)

destroy [**task** *task_list* | **group** *task_group_name*]

The **destroy** subcommand terminates execution of one or more connected processes. By default, the tasks in the current task group (as previously defined by the **group** subcommand) are the ones terminated. You can override this default, however, by specifying a *task_list* or *task_group_name* when you issue the **destroy** subcommand.

When working with a POE application, be aware that terminating any process of the application will cause POE to terminate all of the application's processes. This termination of all processes is a function of POE, not of the PCT. For more information, refer to *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*.

task *task_list*
Specifies the connected tasks to be terminated. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refer to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

For example, to terminate execution of the tasks in the current task group:

```
pct> destroy
```

To terminate task 8:

```
pct> destroy task 8
```

To terminate the tasks in task group *connected*:

```
pct> destroy group connected
```

disconnect subcommand (of the pct command)

disconnect [**task** *task_list* | **group** *task_group_name*]

The **disconnect** subcommand disconnects the PCT from one or more connected processes. Disconnecting from a process removes any performance collection probes from the process. Disconnecting from a process does not terminate the process; the process will continue to run. Once a process is disconnected, the PCT will no longer be able to control execution of, or instrument, the process. By default, the tasks in the current task group (as previously defined by the **group** subcommand) are the ones that are disconnected. You can override this default, however, by specifying a task list or task group name when you issue the **disconnect** subcommand.

task *task_list*
Specifies the connected POE tasks to be disconnected. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

For example, to disconnect from the tasks in the current task group:

```
pct> disconnect
```

To disconnect from task 8:

```
pct> disconnect task 8
```

To disconnect from the tasks in task group *connected*:

```
pct> disconnect group connected
```

exit subcommand (of the pct command)

exit [**destroy**]

The **exit** subcommand exits the PCT. If you loaded the target application, its process(es) will also be terminated. If you merely connected to the target

application, the process(es) will continue to run unless you use the **destroy** clause to explicitly instruct the PCT to kill the connected processes. Since terminating any process of the POE application will cause POE to terminate all of the POE application's processes, the **destroy** clause effectively terminates the entire POE application.

For example, to exit the PCT, but allow all of its connected processes to continue running:

```
pct> exit
```

To exit the PCT and terminate the connected target application processes:

```
pct> exit destroy
```

file subcommand (of the pct command)

file [**task** *task_list* | **group** *task_group_name*] *"regular_expression"*

The **file** subcommand lists, for one or more tasks, any associated source file names that match a regular expression that you supply. By default, this subcommand applies to the current task group (as previously defined by the **group** subcommand). You can override this default, however, by specifying a task list or task group name when you issue the **file** subcommand.

The files are listed by this subcommand as a table with column headings for the task identifier, file identifier, file name, and, if available, the path.

The file identifiers are determined by sorting the files alphabetically and numbering them starting from 0. The path will be shown only if the file path information was supplied when you compiled a file.

task *task_list*
: Specifies the connected POE tasks whose source file names you want to list. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
: Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

"regular_expression"
: An AIX regular expression (file-name substitution pattern) enclosed in quotation marks that identifies the files to list. The **file** subcommand will filter the list of file names using this regular expression; only file names that match this regular expression pattern will be listed.

For example, to list all the files in the current task group:

```
pct> file "*"
Tid File Id File Name Path
--- ------- --------- -------------
0   0       bar.c     ../../lib/src
0   1       foo1.c    ../../lib/src
0   2       foo2.c    ../src
pct>
```

To list only the files in task 0 that begin with the letter "f"

```
pct> file task 0 "f*"
Tid File Id File Name Path
--- ------- --------- -------------
0   1       foo1.c    ../../lib/src
0   2       foo2.c    ../src
pct>
```

find subcommand (of the pct command)

find [**task** *task_list* | **group** *task_group_name*]
function *"regular_expression_to_match_function_name"*

The **find** subcommand lists all function names that match a regular expression pattern that you supply. This subcommand is intended for situations when you wish to instrument a particular function, but do not know which file contains the function.

The function names found are listed by this subcommand as a table with column headings for task identifier, file identifier, file name, and function name.

task *task_list*
: Specifies the connected POE tasks whose source files you want to search. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
: Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

function *"regular_expression_to_match_function_name"*
: An AIX regular expression (file-name substitution pattern) enclosed in quotation marks that identifies the functions to locate. Matching is performed using rules of AIX file-name pattern matching. The **find** subcommand will filter the list of function names using this regular expression; only function names that match this regular expression pattern will be listed.

For example, to list all the functions in task 0 that match the regular expression *comp**:

```
pct> find task 0 function "comp*"
Tid File Id File Name Function Name
--- ------- --------- -------------
0   23      main.c    compute
0   23      main.c    compare
0   25      sort.c    compare2
pct>
```

function subcommand (of the pct command)

function [**task** *task_list* | **group** *task_group_name*]
{**file** *"regular_expression"*[,*"regular_expression"*] | **fileid** *file_identifier*[,*file_identifier*]}...
"regular_expression_to_match_function_name"

The **function** subcommand lists, for one or more tasks, the names of the functions contained in a source file that match a regular expression search pattern you supply. The file whose functions are listed can be specified as a file identifier or as a regular expression that matches the file name. The file information can be ascertained by the **file** subcommand, or, if you are unsure which file the function is located in, the **find** subcommand. By default, this subcommand applies to the current task group (as previously defined by the **group** subcommand). You can override this default, however, by specifying a task list or task group name when you issue the **function** subcommand.

The function names are listed by this subcommand as a table with column headings for task identifier, file identifier, function identifier, file name, and function name.

The function identifiers are determined by sorting the functions contained in a file alphabetically starting from 0. Each file's functions are numbered sequentially starting from 0.

task *task_list*
: Specifies the connected POE tasks containing the source files whose functions you want to list. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
: Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

file *"regular_expression"* **[,***"regular_expression"***]**
: Specifies, using one or more regular expression patterns, the file(s) whose functions you want to list. The regular expression patterns must be contained in quotation marks.

fileid *file_identifier* **[,***file_identifier***]**
: Specifies, using one or more file identifiers as returned by the **file** subcommand, the file(s) whose functions you want to list.

"regular_expression_to_match_function_name"
: A regular expression enclosed in quotation marks that identifies the function names to list. Matching is performed using rules of AIX file-name pattern matching. The **function** subcommand will filter the list of function names using this expression; only function names (for the tasks/file indicated) that match the regular expression will be listed.

For example, to list all the functions in the file "bar.c" in task 0:

```
pct> function task 0 file "bar.c" "*"
Tid File Id Function Id File Name Function Name
--- ------- ----------- --------- -------------
0   1       0           bar.c     func0
0   1       1           bar.c     func1
pct>
```

To list all the functions in the file "bar.c" (using the file identifier) in task 0:

```
pct> function task 0 fileid 1 "*"
Tid File Id Function Id File Name Function Name
--- ------- ----------- --------- -------------
0   1       0           bar.c     func0
0   1       1           bar.c     func1
pct>
```

To list, for task 0, all of the functions in files beginning with ″b″ or ″d″:

```
pct> function task 0 file "b*", "d*" "*"
Tid File Id Function Id File Name Function Name
--- ------- ----------- --------- -------------
0   3       0           bar.c     func0
0   3       1           bar.c     func1
0   3       2           bar2.c    func_xyz
0   4       0           bar2.c    calc
0   4       1           bar2.c    do_math
0   4       2           bar2.c    sum
pct>
```

group subcommand (of the pct command)

group default *task_group_name*

group add *task_group_name task_list*

group delete *task_group_name* [*task_list*]

The **group** subcommand can perform three distinct actions related to task groups:

- Using the **default** action of the **group** command:

  group default *task_group_name*

  you can set the command context on a particular task group. When you do this, the task group you specify becomes the current task group; certain other subcommands that you issue (such as the **file**, **function**, and **point** subcommands) will, by default, apply only to the tasks in the current task group.
- Using the **add** action of the **group** subcommand:

  group add *task_group_name task_list*

  you can create a new task group, or add tasks to an existing task group.
- Using the **delete** action of the **group** subcommand:

  group delete *task_group_name* [*task_list*]

  you can delete, or delete selected tasks from, a task group. If a task list is specified, these tasks are removed from the task group; otherwise, the entire task group is deleted.

In addition to any task groups you create using the **group** subcommand, note that there are two task groups that are created automatically by the PCT when you issue either the **load** or **connect** subcommands. These automatically-created task groups are named *all* and *connected*. The *all* task group contains all tasks in the current application, while the *connected* task group contains the set of tasks to which the PCT is connected.

task_group_name
refers to the name of the task group that, depending on the particular **group** subcommand action you are executing, you want to:

- make the default task group
- create or add tasks to
- delete or remove tasks from

task_list
Refers to the list of tasks that, depending on the particular **group** subcommand action you are executing, you want to either add to, or delete from, the task group. The tasks can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

For example, to create a task group *master* consisting of task 0, and a task group *workers* consisting of tasks 1 through 20.

```
pct> group add master 0
pct> group add workers 1:20
```

To add tasks 21 through 30 to the task group *workers*:

```
pct> group add workers 21:30
```

To make the group *workers* the default task group:

```
pct> group default workers
```

To remove tasks 21 through 30 from the task group *workers*.

```
pct> group delete workers 21:30
```

To delete the task group *workers*:

```
pct> group delete workers
```

help subcommand (of the pct command)

help [*command_name*]

The **help** subcommand can either list all of the PCT's subcommands, or else return the syntax of a particular subcommand.

command_name
refers to the name of the PCT subcommand you want help on.

For example, to get a listing of all of the PCT's subcommands:

```
pct> help
```

To get the syntax of the **load** subcommand:

```
pct> help load
```

list subcommand (of the pct command)

list {[**task** *task_list* | **group** *task_group_name*]
[**file** *"regular_expression"* [,"regular_expression"]... |

fileid *file_identifier*[,*file_identifier*]...] [**line** *line_number_range*]}

list next

The **list** subcommand returns the contents of a file. The first time you issue this subcommand, you should specify a file using the **file** or **fileid** clause. Doing this will list the entire file's contents. To list only a portion of the file's contents, specify a line number range using the **line** clause. To minimize typing, the PCT records the number of the last source code line displayed; issuing the **list next** subcommand will display the next few lines of the source code. By default, this form of the subcommand applies to the current task group (as previously defined by the **group** subcommand). You can override this default, however, by specifying a task list or task group name when you issue the **list** subcommand.

task *task_list*
: Specifies the connected POE tasks containing the source files whose contents you want to list. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
: Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

file *"regular expression"* **[,***"regular expression"***]...**
: Specifies, using one or more regular expressions, the file whose contents you want to list. Only the first file that matches the regular expression(s) will be listed. If this file cannot be located, an error will be returned, regardless of whether a subsequent file match could have been made.

fileid *file_identifier***[,***file_identifier***]...**
: Specifies, using one or more file identifiers as returned by the **file** subcommand, the file(s) whose contents you want to list.

line *line_number_range*
: The line number range of the source code you want to list. Use a colon to separate the ends of the range (for example 1:20).

next displays the next few lines of source code after the range previously returned by the **list** subcommand.

For example, to list lines 1 through 20 of the source file *bar.c*:

```
pct> list file "bar.c" line 1:20
```

To then list the next few lines in *bar.c*:

```
pct> list next
```

load subcommand (of the pct command)

load {{[**poe**] **exec** *absolute_path_to_executable* } | {**poe**
[**mpmdcmd** *path_to_poe_commands_file*] [**poeargs** *"poe_arguments_string"*]}
[**args** *"program_arguments_string"*] [**stdout** *standard_out_file_name*]
[**stderr** *standard_error_file_name*] [**stdin** *standard_input_file_name*]

The **load** subcommand loads a serial or POE application for execution. Once an application is loaded, you can instrument it with probes, or control its execution using the **start**, **suspend**, **resume**, and **destroy** subcommands. The **load** subcommand is intended for applications that are not already executing; to connect to applications that are already executing, use the **connect** subcommand. The **poe** clause indicates that the application is a POE application; if not specified, the **load** subcommand assumes you are loading a serial application. The **load** subcommand loads the application into memory in a ″stopped state″ with execution suspended at its first executable instruction. You can start execution of the application using the **start** subcommand.

poe Specifies that you are loading a POE program.

exec *absolute_path_to_executable*
Specifies the full path to the executable file. If you are loading a POE application, you must also include the keyword **poe** on the command line.

mpmdcmd *path_to_poe_commands_file*
Specifies that the POE program you're loading follows the Multiple Program Multiple Data (MPMD) model and indicates the path to the POE commands file listing the executable programs to run. For more information on POE commands files, refer to the manual *IBM Parallel Environment for AIX: Operation and Use, Volume 1*.

poeargs ″*poe_arguments_string*″
Specifies command-line arguments that are passed to the **poe** command to control various aspects of the Parallel Operating Environment. For a complete listing of the POE arguments you can supply, refer to the manual *IBM Parallel Environment for AIX: Operation and Use, Volume 1*. The POE arguments should be provided as a string delimited by double quotation marks. Embedded quotation marks can be included in the string if each mark is preceded by an escape character (\). Embedded escape characters may also be included if they are preceded by an additional escape character.

args ″*program_arguments_string*″
Specifies command-line arguments that are passed to the application. Note that these are not POE arguments, which are instead specified by using the **poeargs** clause. The program arguments should be provided as a string delimited by double quotation marks. Embedded quotation marks can be included in the string if each mark is preceded by an escape character (\). Embedded escape characters may also be included if they are preceded by an additional escape character.

stdout *standard_out_file_name*
Redirects the target application's standard output to the file specified.

stderr *standard_error_file_name*
Redirects the target application's standard error to the file specified.

stdin *standard_input_file_name*
Reads the target application's standard input from a file.

For example, the following command loads the serial executable *foo* and passes it the argument string ″*a b c*″:

```
pct> load exec /u/example/bin/foo args "a b c"
```

The following command loads the POE executable *parallel_foo* and passes it POE arguments:

```
pct> load poe exec /u/example/bin/parallel_foo poeargs "-procs 4 -hfile /tmp/host.list"
```

The following command loads an MPMD POE program. The executable files to load are listed in the POE commands file */u/example/bin/foo.cmds*:

```
pct> load poe mpmdcmd /u/example/bin/foo.cmds poeargs "-procs 3 -hfile /tmp/host.list"
```

point subcommand (of the pct command)

point [**task** *task_list* | **group** *task_group_name*]
{**file** *"regular_expression"*[,*"regular_expression"*]... |
fileid *file_identifier*[,*file_identifier*]...}
[**function** *"regular_expression"*[,*"regular_expression"*]... |
funcid *function_identifier*[,*function_identifier*]...]

Lists the instrumentation points (at the file or function level) where custom user markers can be added by the **trace add** subcommand. You only need to identify instrumentation points when installing custom user markers using the **trace add** subcommand. You do not need the instrumentation point information if installing MPI trace probes using the **trace add** subcommand or profile probes using the **profile add** subcommand. By default, this subcommand will list the instrumentation points for the tasks in the current task group (as previously defined by the **group** subcommand). You can override this default, however, by specifying a task list or task group name when you issue the **point** subcommand. The **file** or **fileid** clause specifies the file(s) whose instrumentation points you want listed. Using the **function** clause, you can specify one or more functions whose instrumentation points you want listed.

The instrumentation points are listed by this subcommand as a table with headings for task identifier, file identifier, function identifier, point identifier, point type, and callee name.

The point identifiers are determined by numbering the points, starting from 0, according to their location in each function. The first instrumentation point in the function is given the identifier 0, the second is given the identifier 1, and so on. Each function's instrumentation points are numbered separately starting from 0.

task *task_list*
: Specifies the connected POE tasks whose instrumentation points you want to list. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
: Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

file *"regular_expression"***[,***"regular_expression"***]...**
: Specifies, using one or more regular expressions (file name substitution patterns), the file(s) whose instrumentation points you want to list. The regular expression(s) must be contained in quotation marks.

fileid *file_identifier***[,***file_identifier***]...**
: specifies, using one or more file identifiers as returned by the **file** subcommand, the file(s) whose instrumentation points you want to list.

function *"regular_expression"***[,***"regular_expression"***]...**
Specifies, using one or more regular expressions, the function(s) whose instrumentation points you want to list. This regular expression must be contained in quotation marks.

funcid *function_identifier***[,***function_identifier***]...**
Specifies, using one or more function identifiers as returned by the **function** subcommand, the function(s) whose instrumentation points you want to list.

For example, to list all the instrumentation points in task 0 for the file *bar.c*:

```
pct> point task 0 file "bar.c"
Tid File Id Function Id Point Id Point Type Callee Name
--- ------- ----------- -------- ---------- ------------
0   54      0           0        0
0   54      0           1        2          printf
0   54      0           2        3          printf
0   54      0           3        2          MPI_Abort
0   54      0           4        3          MPI_Abort
0   54      0           5        1
0   54      1           0        0
0   54      1           1        2          printf
0   54      1           2        3          printf
0   54      1           3        2          printf
0   54      1           4        3          printf
0   54      1           5        2          MPI_Recv
0   54      1           6        3          MPI_Recv
0   54      1           7        2          consume_data
0   54      1           8        3          consume_data
0   54      1           9        2          printf
0   54      1           10       3          printf
0   54      1           11       1
pct>
```

profile add subcommand (of the pct command)

profile add [**task** *task_list* | **group** *task_group_name*]
{{**profname** *profile_type_name* | **profid** *profile_type_identifier*}
[**groupid** *group_identifier* | **groupname** *group_name*]}...
to {**file** *"regular_expression"*[,*"regular_expression"*]... |
fileid *file_identifier*[,*file_identifier*]...}
[**function** *"regular_expression"*[,*"regular_expression"*]...|
funcid *function_identifier*[,*function_identifier*...]]

The **profile add** subcommand adds one or more probes to collect hardware and operating system profile information. You cannot use this subcommand, or any of the **profile** subcommands, unless you have specified that you are collecting profile data. To specify that you are collecting profile data, issue the **select** subcommand with its **profile** clause:

```
select profile
```

If you add multiple profile probes, be aware that they are considered a single set of probes. When removing profile probes using the **profile remove** subcommand, you will not be able to remove individual probes. Instead, you'll have to remove the entire set of probes.

By default, this subcommand will add the probe(s) to the tasks in the current task group (as previously defined by the **group** subcommand). You can override this default, however, by specifying a task list or task group name when you issue the **profile add** subcommand. Be aware, however, that the set of tasks cannot include

different executables in an MPMD application. For example, if an MPMD application consists of executables *a.out* and *b.out*, then this command cannot be applied to a task group that contains both *a.out* and *b.out* tasks.

task *task_list*
Specifies the connected POE tasks to which you want to add the profile probes. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

profname *profile_type_name*
Specifies, using a probe type name, a profile probe type to add. To list the profile probe type names, use the **profile show** subcommand (with its **probetypes** clause specified):

```
pct> profile show probetypes
```

profid *profile_type_identifier*
Specifies, using a probe type identifier, a profile probe type to add. To list the profile probe type identifiers, use the **profile show** subcommand (with its **probetypes** clause specified):

```
pct> profile show probetypes
```

groupid *group_identifier*
If you are collecting hardware counter information, a profile group identifier indicating the specific hardware counter information you want to collect. To get a list of the profile groups available for your hardware, use the command:

```
pct> profile show probetype hwcount
```

groupname *group_name*
If you are collecting hardware counter information, a profile group name indicating the specific hardware counter information you want to collect. To get a list of the profile groups available for your hardware, use the command:

```
pct> profile show probetype hwcount
```

file *"regular_expression"***[,***"regular_expression"***]...**
Specifies, using one or more regular expressions (file name substitution patterns), the file(s) you wish to instrument with profile probes. The regular expressions must be enclosed in quotation marks.

fileid *file_identifier***[,***file_identifier***]...**
Specifies, using one or more file identifiers as returned by the **file** subcommand, the file(s) you wish to instrument with profile probes.

function *"regular_expression"***[,***"regular_expression"***]...**
Specifies, using one or more regular expressions, the functions you wish to instrument with the profile probes. The regular expression must be enclosed in quotation marks.

funcid *function_identifier***[,***function_identifier***]...**
Specifies, using one or more function identifiers as returned by the **function** subcommand, the functions you wish to instrument with the profile probes.

For example, to add a profile probe to collect wall clock data for the current task group:

```
pct> profile add profname wclock to fileid 5 funcid 3
```

To add a profile probe to collect wall clock data, and hardware data using counter group 2:

```
pct> profile add profname wclock profname hwcount groupid 2 to fileid 3
```

profile remove subcommand (of the pct command)

profile remove probe *probe_index*

The **profile remove** subcommand removes the profile probe set specified by the supplied *probe_index*. A profile probe set consists of one or more probes as previously installed by the **profile add** subcommand. An installed profile probe's *probe_index* can be ascertained by the **profile show** subcommand (with its **probes** clause) as in:

```
pct> profile show probes
```

probe *probe_index*
: Specifies, using a probe index, the profile probe set to be removed. The probe index can be ascertained by issuing the **profile show** subcommand with its **probes** clause.

For example, to remove the profile probe set whose index is 3:

```
pct> profile remove probe 3
```

profile set path subcommand (of the pct command)

profile set path *"path_name/output_file_base_name"*

The **profile set path** subcommand specifies the output location and base name for the profile data files generated by profile probes that you install using the **profile add** subcommand.

"path_name/output_file_base_name"
: specifies a relative or full path to the desired location for the profile output files, followed by the output file base name. The base name is needed because the data collected by the PCT will be saved as a file on each host machine where a connected process with probes is running. The file name will consist of the base name you supply followed by a node-specific suffix supplied by the PCT. If a relative path is specified, note that the location will be relative to the directory where you started the PCT.

For example, to specify the relative path *profile* as the location for profile output files and *output* as the base name:

```
pct> profile set path "profile/output"
```

profile show subcommand (of the pct command)

profile show {**probes** | **probetypes** | **probetype** *probe_type_name* | **path**}

The **profile show** subcommand lists, depending on the clause you specify, either the currently installed profile probes, the list of profile probe types that you can install, the options for a probetype, or the profile file output location.

probes
Specifies that the **profile show** subcommand should list the currently installed profile probes (including the probe index). The probe index information is needed when removing a profile probe using the **profile remove** subcommand.

probetypes
Specifies that the **profile show** subcommand should list the available probe types you can add using the **profile add** subcommand.

probetype *probe_type_name*
Specifies that the subcommand should list the options for the specified probe type. Currently, only the hardware counter probe type has options.

path Specifies that you want the **profile show** subcommand to return the profile file output location and base name as set by the **profile set path** subcommand.

For example, to list the installed profile probes:

```
pct> profile show probes
```

To list available profile probe types:

```
pct> profile show probetypes
Prof Id Prof Name Description
------- --------- ----------------
0       wclock    wall clock
1       rusage    resource usage
2       hwcount   hardware counter
pct>
```

resume subcommand (of the pct command)

resume [**task** *task_list* | **group** *task_group_name*]

The **resume** subcommand resumes execution of one or more processes that have previously been suspended by the **suspend** subcommand. By default, the tasks in the current task group (as previously defined by the **group** subcommand) are the ones that have their execution resumed. You can override this default, however, by specifying a task list or task group name when you issue the **resume** subcommand.

task *task_list*
Specifies the connected POE tasks that you want to resume executing. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

For example, to resume execution of all tasks in the current task group:

```
pct> resume
```

To resume execution of tasks 0 through 20:

```
pct> resume task 0:20
```

To resume execution of the tasks in task group *mygroup*:

```
pct> resume group mygroup
```

run subcommand (of the pct command)

run *"pct_script_file"*

The **run** subcommand executes a series of PCT commands that are stored in a "PCT script file". A PCT script file is an ASCII file that lists a sequence of PCT subcommands. Each PCT subcommand is placed on a separate line in the PCT script file. Lines beginning with a # (pound sign) character are comments and will not be executed by the PCT.

"pct_script_file"
Specifies the name of the PCT script file whose subcommands you want to execute. The file name must be enclosed in quotation marks.

For example, to execute the PCT subcommands contained in the PCT script file *myscript.cmd*:

```
pct> run "myscript.cmd"
```

select subcommand (of the pct command)

select {**trace** | **profile**}

The **select** subcommand enables you to select the type of probe data you will be collecting.

trace Specifies that you intend to collect MPI or custom user event traces for eventual analysis using Jumpshot or the **utestats** utility.

profile
Specifies that you intend to collect hardware and operating system profiles for analysis using the Profile Visualization Tool.

For example, if you will be adding trace probes (using the **trace add** subcommand) for collecting MPI or custom user event data:

```
pct> select trace
```

If, on the other hand, you will be adding profile probes (using the **profile add** subcommand) for collecting hardware and operating system profiles:

```
pct> select profile
```

set subcommand (of the pct command)

set sourcepath [**relative**] *"path_list"*

The **set** subcommand enables you to set the path used when displaying the contents of a file using the **list** subcommand. The initial value for the source path is the directory in which the tool was started.

relative
Specifies that, if relative path information is included as part of the file name supplied to the **list** subcommand, the relative path should be used together with the directories listed in the *pathlist*.

For example, say one of the source files in the application is named *"../../myapp/src/compute.c"* and the source path is *"/tmp:/usr/tmp:/home/mydir/examples/yourapp"*. If the **relative** keyword is used when setting the source path, the PCT searches the following directories when the **list /../../myapp/src/compute.c** subcommand is issued.

```
/tmp/../../myapp/src/
/usr/tmp/../../myapp/src/
/home/mydir/examples/yourapp/../../myapp/src/
```

If the **relative** keyword is not used when setting the source path, however, the following directories are searched:

```
/tmp/
/usr/tmp/
/home/mydir/examples/yourapp/
```

"path_list"
A colon-delimited list that specifies the path the **list** subcommand will use to search for source files.

show subcommand (of the pct command)

show { events | **group** *task_group_name* |
groups | **points** | **ps** | **sourcepath** |
tools }

The **show** subcommand returns, depending on the form of the subcommand you use, various information about the target application and the PCT.

- Using the form **show events** returns a list of the possible events that, if you place the PCT in an event loop using the **wait** subcommand, can break the PCT out of the loop. Be aware that the **wait** subcommand is intended only for use within scripts you write, and is not intended for interactive command-line sessions.
- Using the form:

  show group *task_group_name*

  returns, for each task in the specified task group, the task identifier, the program name, the name of the host machine on which the task is running, the CPU type, and the task state.
- Using the form:

  show groups

  returns a list of task groups. This includes any task groups created by default (the task groups *all* and *connected*), and any task groups you created using the **group** subcommand. An ampersand character (@) is displayed to the right of the default task group.
- Using the form:

  show points

  returns a list of the available instrumentation point types. This enables you to understand the numeric point type returned by the **point** subcommand.
- Using the form:

show ps

returns a list of the processes you own on the node where you started the PCT. This information is needed when connecting to an application using the **connect** subcommand.

- Using the form:

  show sourcepath

  returns a list of directories searched when displaying the contents of a file using the **list** subcommand. You can set the source path using the **set** subcommand.

- Using the form:

  show tools

  returns a list of the types of information you can collect using the PCT (for this release, ″trace″ and ″profile″). This information is needed when selecting the type of data you will be collecting using the **select** subcommand.

For example, to show the tasks in the current task group:

```
pct> show group
```

To show the tasks in the task group ″connected″:

```
pct> show group connected
```

To show the processes that you own on the host machine:

```
pct> show ps
```

start subcommand (of the pct command)

start

The **start** subcommand starts execution of an application you have loaded using the **load** subcommand. (The **load** subcommand loads an application into memory in a ″stopped state″ with execution suspended at the first executable instruction.)

For example, to start execution of the currently-loaded application:

```
pct> start
```

stdin subcommand (of the pct command)

stdin [{″*string*″ | **eof**}]

The **stdin** subcommand sends the supplied string as standard input to the currently loaded application. If no string is supplied, the **stdin** subcommand will send a newline character to the application. If the **eof** option is supplied, the **stdin** subcommand will send an end-of-file character to the application.

Be aware that this subcommand is intended only for applications that you have loaded using the **load** subcommand. If you have instead connected to an application using the **connect** subcommand, you cannot send standard input text using the **stdin** subcommand.

Also be aware that you can, when loading an application using the **load** subcommand, indicate that the application should read standard input from a file specified by the **stdin** option. If the **stdin** option is used when loading an application with the **load** subcommand, note that the **stdin** subcommand cannot be used.

"string"
Specifies a text string to send to standard input. The string should be enclosed in quotes, and embedded formatting characters (such as \n) are permitted. If no string is supplied, the **stdin** subcommand will send a newline character to the application.

eof sends an end-of-file character to the input stream reading this input data.

For example:

```
pct> stdin "now is the time \nfor all good men"
```

suspend subcommand (of the pct command)

suspend [**task** *task_list* | **group** *task_group_name*]

The **suspend** subcommand suspends execution of one or more processes. By default, the tasks in the current task group (as previously defined by the **group** subcommand) are the ones that are suspended. You can override the default, however, by specifying a task list or task group name when you issue the **suspend** subcommand. You can resume execution of tasks suspended by this subcommand by issuing the **resume** subcommand.

task *task_list*
Specifies the connected POE tasks that you want to suspend. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

For example, to suspend execution of all tasks in the current task group:

```
pct> suspend
```

To suspend execution of tasks 0 through 20:

```
pct> suspend task 0:20
```

To suspend execution of the tasks in task group *"mygroup"*:

```
pct> suspend group mygroup
```

trace add subcommand (of the pct command)

trace add [**task** *task_list* | **group** *task_group_name*]
{**mpiid** *probetype_number_list* | **mpiname** *probe_name_list*} **to**
{**file** *"regular_expression"*[,*"regular_expression"*]... |
fileid *file_identifier*[,*file_identifier*]... }
[**function** *"regular_expression"*[,*"regular_expression"*]... |

funcid *function_identifier*[,*function_identifier*]...]

trace add [**task** *task_list* | **group** *task_group_name*]
{**simplemarker** *"marker_name"* |
{{**beginmarker** | **endmarker**} *"marker_name"*}
| {**traceon** | **traceoff**}} **to** {**file** *"regular_expression"*[,*"regular_expression"*]... |
fileid *file_identifier*[,*file_identifier*]... }
{**function** *"regular_expression"*[,*"regular_expression"*]... |
funcid *function_identifier*[,*function_identifier*]...} **pointid** *point_identifier*

The **trace add** subcommand enables you to add the following types of probes to one or more tasks. You can add:

- MPI trace probes. If you add multiple MPI trace probes, be aware that they are considered a single set of probes. When removing MPI trace probes using the **trace remove** subcommand, you will not be able to remove selected probes. Instead, you'll have to remove the entire set of probes.
- simple user markers to trace events of interest
- begin user markers and end user markers to trace intervals of interest
- user markers to force tracing on and off

You cannot use this subcommand, or any of the trace subcommands, unless you have specified that you are collecting trace data. To specify that you are collecting trace data, issue the **select** subcommand with its **trace** clause:

```
pct> select trace
```

You also need to specify the output location and a ″base name″ prefix for the trace files. To do this, use the **trace set path** command. For example:

```
pct> trace set path "/home/timf/tracefiles/mytrace"
```

By default, this subcommand will add the probes to the tasks in the current task group (as previously defined by the **group** subcommand). You can override this default, however, by specifying a task list or task group name when you issue the **trace add** subcommand. Be aware, however, that the set of tasks cannot include different executables in an MPMD application. For example, if an MPMD application consists of executables *a.out* and *b.out*, then this command cannot be applied to a task group that contains both *a.out* and *b.out* tasks.

task *task_list*
: Specifies the connected POE tasks to which you want to add the trace probes or user markers. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
: Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

mpiid *probetype_number_list*
: A probe identifier (or a list of comma-separated probe identifiers) indicating the type of MPI data (collective communication, point-to-point communication, one-sided operations, and so on) that you want to collect. To get a list of the probe identifiers, issue the **trace show** subcommand with its **probetypes** clause as in:

```
pct> trace show probetypes
```

mpiname *probe_name_list*
A probe name (or a list of comma-separated probe names) indicating the type of MPI data (collective communication, point-to-point communication, one-sided operations, and so on) that you want to collect. To get a list of the probe names, issue the **trace show** subcommand with its **probetypes** clause as in:

```
pct> trace show probetypes
```

simplemarker *"marker_name"*
Indicates that the probe is a simple marker being placed in the target application to trace a particular event of interest. A simple marker appears in the trace record as a single point.

{beginmarker | endmarker} *"marker_name"*
Specifies that the probe is a user marker that marks either the beginning or ending of a named user state. You need to mark both the beginning and ending of the range with the same *"marker_name"* (a string that will be used to identify the user state in the trace record). You can only use a particular marker name for one begin marker/end marker pair. The state will appear in the trace record as a region.

{traceon | traceoff}
Specifies that the probe is a user marker that will either force tracing on or off. This provides a finer degree of trace control than is otherwise available when merely specifying the file and function to trace.

file *"regular_expression"***[,***"regular_expression"***]...**
Specifies, using one or more regular expressions (file name substitution patterns), the file(s) you wish to instrument. The regular expression must be contained in quotation marks.

fileid *file_identifier***[,***file_identifier***]...**
Specifies, using one or more file identifiers as returned by the **file** subcommand, the files you wish to instrument.

function *"regular_expression"***[,***"regular_expression"***]...**
Specifies, using one or more regular expressions, the function(s) you want to instrument.

funcid *function_identifier***[,***function_identifier***]...**
Specifies, using one or more function identifiers as returned by the **function** subcommand, the function you want to instrument.

pointid *point_identifier*
Specifies, using a point identifier, the instrumentation point at which to add the user markers.

For example, to trace all MPI events in the file *"bar.c"*:

```
pct> trace add mpiname all to file "bar.c"
```

To add a begin state marker named *"green"* to the second point of the first function of file *"foo.c"*:

```
pct> trace add beginmarker "green" to file "foo.c" funcid 0 pointid 1
```

trace remove subcommand (of the pct command)

trace remove {**marker** *marker_id* | **probe** *probe_index*}

The **trace remove** subcommand enables you to remove a custom user marker or a trace probe set.

marker *marker_id*
Specifies the marker identifier of the custom user marker you want to remove. To ascertain the marker identifier, use the **trace show** subcommand with its **markers** clause.

```
pct> trace show markers
```

probe *probe_index*
Specifies, using a probe index, the trace probe set you wish to remove. A trace probe set consists of one or more probes previously installed by the **trace add** subcommand. To ascertain the trace probe set you wish to remove, use the **trace show** subcommand with its **probes** clause as in:

```
pct> trace show probes
```

For example, to remove the trace probe whose probe identifier is ″2″:

```
pct> trace remove probe 2
```

trace set subcommand (of the pct command)

trace set { **path** *″path_name/output_file_base_name″* | [**bufsize** *buffer_size*] [{**event** {**mpi** | **process** | **idle**} | {**event** [**mpi,**] [**process,**] [**idle**]}] [**logsize** *maximum_log_size*]}

The **trace set** subcommand enables you to specify various settings for event trace collection. You cannot use this subcommand, or any of the trace subcommands, unless you have specified that you are collecting trace data. To specify that you are collecting trace data, issue the **select** subcommand with its **trace** clause:

```
select trace
```

The settings you make with this subcommand will stay in effect until you issue the **select** subcommand.

path *″path_name/output_file_base_name″*
Specifies a relative or full path name to the desired location for trace files followed by the output file base name. The base name is needed because the data collected by the PCT will be stored as a file on each host machine where a connected process with probes is running, The file name will consist of the base name you supply followed by a node specific suffix supplied by the PCT.

bufsize *buffer_size*
Specifies the AIX trace buffer size in Kilobytes. This value can be at most 1024, which is also the default value.

[{event {mpi | process | idle} | {event [mpi,] [process,] [idle]}]
Specifies the type of events (MPI events, process dispatch events, and CPU idle events) that are traced. By default, MPI events and process dispatch events are traced. Tracing process dispatch events and CPU idle events can result in larger trace files, but the additional information can provide useful context for the MPI information collected.

If you want to specify more than one event type, use a comma to separate the event type names.

logsize *maximum_log_size*
Specifies the maximum trace file size in Megabytes. The default is 20 M.

For example, to specify the directory tracefiles/mytrace as the output directory for the trace files:

```
pct> trace set path "tracefiles/mytrace"
```

To specify the buffer size to be 900 K:

```
pct> trace set bufsize 900
```

To specify the maximum trace file size to be 25 M:

```
pct> trace set logsize 25
```

To specify that CPU idle events should be collected:

```
pct> trace set event idle
```

To specify that MPI and CPU idle events should be collected:

```
pct> trace set event mpi, idle
```

trace show subcommand (of the pct command)

trace show {[**task** *task_list* | **group** *task_group_name*] {**markers** | **probes**} | **probetypes** | **path**}

The **trace show** subcommand lists, depending on the clause you specify, either:

- the currently installed trace probes:

  trace show [**task** *task_list* | **group** *task_group_name*] **probes**
- the currently installed user markers:

  trace show [**task** *task_list* | **group** *task_group_name*] **markers**
- the list of available probe types you can add using the **trace add** subcommand:

  trace show probetypes
- the trace file output location and base name (as set by the **trace set path** subcommand):

  trace show path

When listing the currently installed trace probes or user markers, the action is performed for the tasks in the current task group (as previously defined by the **group** subcommand). You can override this default, however, by specifying a task list or task group name when you issue the **trace show** subcommand.

task *task_list*
: Specifies the connected POE tasks whose trace probes or user markers you want to list. The tasks in the POE application can be specified by listing individual values separated by commas (1,3,8,9), by giving a range of tasks using a colon to separate the ends of the range (12:15 refers to tasks 12, 13, 14, and 15), by giving a range and increment value using colons to separate the range and increment values (20:26:2 refers to tasks 20, 22, 24, and 26), or by using a combination of these (12:18,22,30).

group *task_group_name*
: Specifies the name of a task group. Refer to the **group** subcommand for information on creating task groups.

markers
Specifies that you want the **trace show** subcommand to list the currently installed user markers.

probes
Specifies that you want the **trace show** subcommand to list the currently installed trace probes.

probetypes
Specifies that you want the **trace show** subcommand to list the available trace probe types you can add using the **trace add** subcommand.

path Specifies that you want the **trace show** subcommand to return the trace file output location and base name as set by the **trace set path** subcommand.

For example, to list the trace probes installed in the tasks in the current task group:

```
pct> trace show probes
```

To list the user markers for the tasks in the task group ″workers″:

```
pct> trace show markers
```

To list the available probe types:

```
pct> trace show probetypes
```

wait subcommand (of the pct command)

wait

The **wait** subcommand blocks the PCT's execution so that it can wait for asynchronous system events (such as a task terminating) to occur. When one of these asynchronous events occurs, the PCT resumes execution, and returns the event that occurred. Be aware that this command is intended only for use within scripts you write, and is not intended for interactive command-line sessions. If you use it during an interactive command-line session, the only way to break out of the loop is to press **<control>-C** which will kill the PCT.

To see a list of the possible events that can resume execution of the PCT, issue the subcommand:

```
pct> show events
```

For example, the following example blocks execution of the PCT. Execution of the PCT resumes when the target application terminates. The PCT returns the event name ″*app_term*″:

```
pct> wait
app_term
```

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_.

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 *AIX 5L Version 5.1 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 16.

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_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 3 on page 5 and "Reading subcommands from a command file" on page 29.

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), **poe**(1)

Subcommands of the pdbx command

alias subcommand (of the pdbx command)

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.

assign subcommand (of the pdbx command)

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
```

attach subcommand (of the pdbx command)

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.

attribute subcommand (of the pdbx command)

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.

back subcommand (of the pdbx command)

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.

call subcommand (of the pdbx command)

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.

case subcommand (of the pdbx command)

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.

catch subcommand (of the pdbx command)

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.

condition subcommand (of the pdbx command)

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.

cont subcommand (of the pdbx command)

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.

dbx subcommand (of the pdbx command)

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 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.

delete subcommand (of the pdbx command)

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.

detach subcommand (of the pdbx command)

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.

dhelp subcommand (of the pdbx command)

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.

display memory subcommand (of the pdbx command)

<*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

down subcommand (of the pdbx command)

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.

dump subcommand (of the pdbx command)

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.

file subcommand (of the pdbx command)

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.

func subcommand (of the pdbx command)

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.

goto subcommand (of the pdbx command)

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.

gotoi subcommand (of the pdbx command)

gotoi *address*

The **gotoi** subcommand changes the program counter address to the address specified by the *address* parameter.

group subcommand (of the pdbx command)

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.

halt subcommand (of the pdbx command)

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 application. 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.

help subcommand (of the pdbx command)

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

hook subcommand (of the pdbx command)

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.

ignore subcommand (of the pdbx command)

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.

list subcommand (of the pdbx command)

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.

listi subcommand (of the pdbx command)

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.

load subcommand (of the pdbx command)

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
```

map subcommand (of the pdbx command)

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.

mutex subcommand (of the pdbx command)

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 (nonrecursive), 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.

next subcommand (of the pdbx command)

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.

nexti subcommand (of the pdbx command)

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.

on subcommand (of the pdbx command)

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 16 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.

print subcommand (of the pdbx command)

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.

quit subcommand (of the pdbx command)

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
```

registers subcommand (of the pdbx command)

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.

return subcommand (of the pdbx command)

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.

search subcommand (of the pdbx command)

/<*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.

set subcommand (of the pdbx command)

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.

sh subcommand (of the pdbx command)

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
```

skip subcommand (of the pdbx command)

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.

source subcommand (of the pdbx command)

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.

status subcommand (of the pdbx command)

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.

step subcommand (of the pdbx command)

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.

stepi subcommand (of the pdbx command)

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.

stop subcommand (of the pdbx command)

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 29.

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 29.

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.

tasks subcommand (of the pdbx command)

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.

thread subcommand (of the pdbx command)

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.

trace subcommand (of the pdbx command)

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 29.

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 29.

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.

unalias subcommand (of the pdbx command)

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.

unhook subcommand (of the pdbx command)

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.

unset subcommand (of the pdbx command)

unset *name*

The **unset** subcommand removes the set variable associated with the specified *name*.

Related to this subcommand is the **set** subcommand.

up subcommand (of the pdbx command)

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.

use subcommand (of the pdbx command)

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.

whatis subcommand (of the pdbx command)

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.

where subcommand (of the pdbx command)

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.

whereis subcommand (of the pdbx command)

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.

which subcommand (of the pdbx command)

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.

pvt

NAME

pvt – Invokes the Profile Visualization Tool (PVT) in either its graphical-user-interface or command-line mode.

SYNOPSIS

pvt [**-c** [*one_or_more_file_names*]]

The **pvt** command starts the PVT in either its graphical-user-interface mode, or, if the **-c** flag is specified, its command-line mode. In either mode, you can specify one or more file names to start the PVT with profile data showing.

FLAGS

-c Specifies that the PVT should be started in command-line mode. Refer to "Using the Profile Visualization Tool's Command Line Interface" on page 136 for information on the subcommands you can issue once the PVT is running in this mode.

DESCRIPTION

The PVT is a post-mortem analysis tool. It is designed to process profile data files generated by the PCT used in application profiling. You can run the PVT in either its graphical-user-interface mode, or, if the **-cmd** flag is specified, its command-line mode. After processing profile data, you can view the results in the PVT's graphical user interface display, outputted to report files, or saved to a summary file. The PVT provides a command-line interface to process individual profile files directly into a summary file without initializing the graphic display. The command-line interface also enables you to generate textual profile reports.

The **pvt** command's subcommands (for controlling the PVT in command-line mode) are listed alphabetically under "Subcommands of the pvt command" on page 199.

EXAMPLES

To start the PVT in graphical-user-interface mode showing an empty graphical user interface:

```
pvt
```

To start the PVT in graphical-user-interface mode with profile data showing:

```
pvt one_or_ more_file_names
```

To start the PVT in command-line mode:

```
pvt -c
```

To start the PVT in command-line mode with profile data showing:

```
pvt -c one_or_more_file_names
```

RELATED INFORMATION

Commands: **pct**(1)

Subcommands of the pvt command

exit subcommand (of the pvt command)

exit

The **exit** subcommand ends the command line session.

export subcommand (of the pvt command)

export *output_file_name*

The **export** subcommand allows you to export profile data to a specified file. The suffix *.txt* will be appended to the specified file name.

The currently loaded profile data is written to the user-specified file in plain text format, so the data can be loaded easily into a spreadsheet tool, like Lotus 1–2–3. The data that is loaded into the tool can be grouped into the following types of records:

- Profile-session records associated with each process
- Individual function or thread records
- Function statistics records.

load subcommand (of the pvt command)

load *one_or_more_file_names*

The load subcommand loads a set of profile data files into the session. If a set of data already exists, then the existing data is discarded and the newly loaded data becomes the current data to be used in future actions.

report subcommand (of the pvt command)

report [**list** | *output_file_name* | *"one_or_more_report_names" output_file_name* | *"one_or_more_report_ids" output_file_name*]

The report subcommand generates textual reports on the profile data. To show a list of available report types, enter:

```
report list
```

The result of the command will look something like:

- **[0] call_count:** function call count report
- **[1] wclock:** wall clock timer report
- **[2] ru_cpu:** CPU usage reports
- **[3] ru_mem:** memory usage report
- **[4] ru_paging:** paging activities reports
- **[5] ru_cswitch:** context switch activities reports
- **[6] pmc_cycle:** instructions per cycle hardware counter reports
- **[7] pmc_fpu:** floating point hardware counter reports
- **[8] pmc_fxu:** fixed-point hardware counter reports
- **[9] pmc_branch:** branch hardware counter reports

- **[10] pmc_lsu:** load and store hardware counter reports
- **[11] pmc_cache:** cache hardware counter reports
- **[12] pmc_misc:** miscellaneous hardware counter reports

To generate all the available reports to a file, enter:

```
report output_file_name
```

To generate reports by report name, enter:

```
report "one_or_more_report_names" output_file_name
```

For example:

```
report "wclock,ru_cpu" output
```

To generate reports by report id, enter:

```
report "one_or_more_report_ids" output_file_name
```

For example:

```
report "1,2" output
```

The report names or report ids in double quotes must be separated by a comma with no blank space in between. No matter how many reports are selected in one report command, all the reports are outputted to a single file specified in the report command.

sum subcommand (of the pvt command)

sum *summary_file_name*

The sum subcommand creates a summary file of all the loaded data. The merged summary data is written to the file specified in the command.

slogmerge

NAME

slogmerge – Merges multiple UTE interval files into a single SLOG file.

SYNOPSIS

slogmerge [**-?**] [**-n** *number_of_files*] [**-c** *number_of_bytes_per_frame*] [**-o** *output_file_name*] [**-s** *range*] [**-m** *number_of_available_markers*] [**-r** *factor*] [**-g**] *input_file_name_prefix*

The **slogmerge** command merges multiple UTE interval trace files (whose names begin with the *input_file_name_prefix*) into a single SLOG file. The *input_file_name_prefix* must be the last item on the command line.

FLAGS

-? Prints out the usage information for the **slogmerge** command instead of performing the actual merge.

-n *number_of_files*
Specifies the number of input UTE interval files to be merged. The default value is 1.

-c *number_of_bytes_per_frame*
Specifies the number of bytes per frame. The default is 128K bytes.

-o *output_file_name*
Specifies the name for the output file — the merged SLOG file. The **slogmerge** utility will create a file with a .slog extension. If you do not specify an output file name, the default value is *trcfile.slog* in the current directory.

-s *range*
Specifies a list of MPI tasks to be merged. The task IDs in the list can be separated by either a comma (,) or a hyphen (-). If used, the hyphen represents a range of tasks. For example, `-s 0,2,4,5-7` indicates that the user wants to merge threads with MPI task IDs 0, 2, 4, 5, 6, and 7. By default, all tasks/threads in all UTE interval files will be merged.

-m *number_of_available_markers*
Specifies the number of spaces to reserve for user markers in the SLOG interval table. The number of available markers should not be less than the actual number of uniquely named user markers in the UTE trace file, or the **slogmerge** utility will quit. The default number of available markers is 20.

-r *factor*
specifies the factor by which spaces for ″pseudo records″ are reserved. The number of reserved slots for pseudo records is the number of threads in the trace file times the *factor*. If not specified, the default is 2.

Pseudo records are SLOG-specific interval records that are duplicates of certain internal records for visualization purposes. The number of pseudo records could be fairly high, depending on the number of nested states and their time span, and the number of internal records crossing SLOG frame boundaries in the trace. If the number of created pseudo records is more than the reserved slots during the merge process, the **slogmerge** utility will quit. If this happens, you should specify a larger number for this option to reserve more slots for pseudo records.

-g Merge interval files without using global clock records. This is needed when processing interval files generated on nodes with no SP switch.

DESCRIPTION

The **slogmerge** command merges multiple UTE interval trace files into a single SLOG file. A number (as indicated by the **-n** option) of UTE files beginning with the *input_file_name_prefix* will be merged into an output file. The name of this output file is the one specified by the **-o** option, or, if the **-o** option is not specified, the file *trcfile.ute* in the current directory by default. The *input_file_name_prefix* must be the last item in the command line.

ENVIRONMENT VARIABLES

UTEPROFILE

Specifies the name of the file description profile. If not set, the file */usr/lpp/ppe.perf/etc/profile.ute* is the default description profile. This variable is intended for use by IBM support personnel.

EXAMPLES

To merge 5 UTE interval trace files that begin with the prefix *mytrace* into a single SLOG file:

```
slogmerge -n 5 mytrace
```

The above example will create an SLOG file with the default output file name *trcfile.ute*. To specify your own output file name, use the **-o** option.

```
slogmerge -n 5 -o mergedtrc.ute mytrace
```

To additionally specify that only the MPI tasks 2, 4, and 6 through 9 should be merged into the SLOG file, use the **-s** option.

```
slogmerge -n 5 -o mergedtrc.ute -s 2,4,6-9 mytrace
```

FILES

profile.ute default description profile

RELATED INFORMATION

Commands: **uteconvert**(1), **utemerge**(1), **utestats**(1)

uteconvert

NAME

uteconvert – Converts AIX event trace files into UTE internal trace files.

SYNOPSIS

uteconvert [**-?**] [**-n** *number_of_files*]
[**-o** {*output_file_name* | *output_file_name_prefix*}] [**-r**]
{*input_file_name* | *input_file_name_prefix*}

The **uteconvert** command converts one or more AIX event trace files into one or more UTE interval trace files. The *input_file_name* (for converting a single AIX event trace file) or *input_file_name_prefix* (for converting multiple AIX event trace files) must be the last item on the command line.

FLAGS

-? Prints out usage information for the **uteconvert** command instead of converting AIX trace files.

-n *number_of_files*
Specifies the number of AIX event trace files to be converted. If not specified, the default is 1.

-o {*output_file_name* **|** *output_file_name_prefix***}**
If the **-n** option specifies the number of files as 1 (the default), the **-o** option specifies the name of the resulting UTE interval file.

If the **-n** option specifies the number of files as greater than 1, the **-o** option specifies the file name prefix for the resulting UTE interval files. The names of the output files are formed by concatenating the given prefix with a node identifier, starting from 0.

-r removes AIX trace files after they have been processed.

DESCRIPTION

The **uteconvert** command converts one or more AIX event trace files into one or more UTE interval trace files. If the **-n** option specifies the number of files to be converted as 1 (the default), then you supply a single *input_file_name* to the **uteconvert** subcommand. If instead, the **-n** option specifies the number of files to be converted as greater than 1, then an *input_file_name_prefix* is supplied. The *input_file_name* or *input_file_name_prefix* must be the last item on the command line.

ENVIRONMENT VARIABLES

UTEPROFILE
Specifies the name of the file description profile. If not set, the file *profile.ute* in the current directory is the default description profile. This variable is intended for use by IBM support personnel.

EXAMPLES

To convert the AIX trace file *mytrace* into a UTE interval trace file:

```
uteconvert mytrace
```

To convert five trace files with the prefix *mytaces* into UTE interval trace files:

```
uteconvert -n 5 mytraces
```

FILES

profile.ute default description profile.

RELATED INFORMATION

Commands: **slogmerge**(1), **utemerge**(1), **utestats**(1)

utemerge

NAME

utemerge – Merges multiple UTE interval files into a single UTE interval file.

SYNOPSIS

utemerge [**-?**] [**-n** *number_of_files*] [**-o** *output_file_name*] [**-s** *range*] [**-g**] *input_file_name_prefix*

The **utemerge** command merges multiple UTE interval trace files (whose names begin with the *input_file_name_prefix*) into a single UTE file. The *input_file_name_prefix* must be the last item on the command line.

FLAGS

-? Prints out the usage information for the **utemerge** command instead of performing the actual merge.

-n *number_of_files*
Specifies the number of input UTE interval files to be merged. The default value is 1.

-o *output_file_name*
Specifies the name for the output file — the merged UTE file. If not specified, the default value is *trcfile.ute* in the current directory.

-s *range*
Specifies a list of MPI tasks to be merged. The task IDs in the list can be separated by either a comma (,) or a hyphen (-). If used, the hyphen represents a range of tasks. For example, `-s 0,2,4,5-7` indicates that the user wants to merge threads with MPI task IDs 0, 2, 4, 5, 6, and 7. By default, all tasks/threads in all UTE interval files will be merged.

-g Merges interval files without using global clock results. This is needed when processing interval files generated on nodes with no SP switch.

DESCRIPTION

The **utemerge** command merges multiple UTE interval trace files into a single UTE interval trace file. A number (as indicated by the **-n** option) of UTE files beginning with the *input_file_name_prefix* will be merged into an output file. The name of this output file is the one specified by the **-o** option, or, if the **-o** option is not specified, the file *trcfile.ute* in the current directory by default. The *input_file_name_prefix* must be the last item in the command line.

ENVIRONMENT VARIABLES

UTEPROFILE
Specifies the name of the file description profile. If not set, the file */usr/lpp/ppe.perf/etc/profile.ute* is the default description profile. This variable is intended for use by IBM support personnel.

EXAMPLES

To merge 5 UTE interval trace files that begin with the prefix *mytrace* into a single UTE file:

```
utemerge -n 5 mytrace
```

The above example will create a UTE file with the default output file name *trcfile.ute*. To specify your own output file name, use the **-o** option.

```
utemerge -n 5 -o mergedtrc.ute mytrace
```

To additionally specify that only the MPI tasks 2, 4, and 6 through 9 should be merged into the UTE file, use the **-s** option.

```
utemerge -n 5 -o mergedtrc.ute -s 2,4,6-9 mytrace
```

FILES

profile.ute default description profile

RELATED INFORMATION

Commands: **uteconvert**(1), **slogmerge**(1), **utestats**(1)

utestats

NAME

utestats – Generates statistics tables from UTE interval files.

SYNOPSIS

utestats [**-?**] [**-o** *output_file_name*]
[**-B** *number_of_bins*] *input_file* [*input_file*]...

The **utestats** command generates statistics tables from one or more UTE interval file. By default, six two-dimensional tables are generated. These tables are:

- Time Bin vs. Node
- Thread vs. Event Type
- Event Type vs. Thread
- Node vs. Event Type
- Event Type vs. Node
- Node vs. Processor

The computed statistic for all tables is the sum of the duration. By default, the statistics tables will be written to standard output. You can optionally save the statistics tables to a file using the **-o** flag.

FLAGS

-? Prints out the usage information for the **utestats** command instead of generating statistics tables.

-o *output_file_name*
Specifies the name of a file to which the statistics tables will be saved. If not specified, the statistics tables will be written to standard output.

-B *number_of_bins*
Specifies the number of bins in the Time vs. Node table. The default is 50.

DESCRIPTION

The **utestats** utility is able to take individual UTE interval files or a merged UTE interval file as input. If a number of individual UTE interval files are specified, the timestamps in each file will start at 0 without alignment with respect to global clock values. If, instead, a merged UTE interval file is specified, the timestamps of records from different nodes will already have been adjusted with respect to the global clock value.

By default, six two-dimensional tables are generated. These tables are:

- Time Bin vs. Node
- Thread vs. Event Type
- Event Type vs. Thread (a row/column transposition of the Thread vs. Event Type table)
- Node vs. Event Type
- Event Type vs. Node (a row/column transposition of the Node vs. Event Type table)
- Node vs. Processor

The computed statistic for all the tables is the sum or the duration. As you can see, several tables are simply row/column transpositions of other tables. These transposed tables are provided so that a program used to visualize the tables does not have to transpose a table in order to show a transposed view.

The output of the **utestats** command is written in tab-separated-value format; each line of output is a row of a table, and columns in a row are separated by a tab character. Tables are separated by a Form Feed character (0x0c). This format is used to make it easy to import a **utestats** output file into a spreadsheet program.

A Node vs. Processor table would look like the following (where the tabs have been replaced by spaces to make the column alignment clearer).

```
node/cpu            0               1
   0             2.823739        2.258315
   1             0.873746        4.241253
   2             0.956515        4.322891
   3             0.853188        4.334650
```

The first value ″node/cpu″ is the name of the table. It consists of the row title followed by a ″/″ followed by a column title. This table contains statistics aggregated over interval records whose field values for ″node″ and ″cpu″ are the same. The values ″node″ and ″cpu″ are the field names as stored in the UTE profile file. The rest of the values in the first row are the column labels; these are the values that appeared in the ″cpu″ field in at least one interval record.

With other rows, the first field is the row label; it is a value that appeared in the node field in at least one interval record. The other fields in a row are the accumulated duration of all interval records with the same (″node″, ″cpu″) pair of values. For example, the accumulated duration of all interval records for ″cpu″ 1 of ″node″ 0 was 2.258315 seconds.

ENVIRONMENT VARIABLES

UTEPROFILE

Specifies the name of the file description profile. If not set, the file *profile.ute* in the current directory is the default description profile. This variable is intended for use by IBM support personnel.

EXAMPLES

To generate statistics tables for a single UTE interval file:

```
utestats mytrace.ute
```

The above example will write the statistics tables to standard output. To redirect the output to a file, use the **-o** option.

```
utestats -o stattables mytrace.ute
```

You can also specify multiple UTE interval files from which statistics should be generated.

```
utestats mytrace.ute mytrace2.ute mytrace3.ute
```

FILES

profile.ute default description profile

RELATED INFORMATION

Commands: **uteconvert**(1), **utemerge**(1)

xprofiler

NAME

xprofiler – Invokes the Xprofiler, a GUI-based performance profiling tool.

SYNOPSIS

xprofiler [*program*] [**-b**] [**-h**]
[**-s**] [**-z**] [**-a**] [**-c**]
[**-L** *pathname*]
[[**-e** *name*]...]
[[**-E** *name*]...]
[[**-f** *name*]...]
[[**-F** *name*]...]
[**-disp_max** *number_of_functions*]
[[*gmon.out*]...]

The **xprofiler** command invokes the Xprofiler, a GUI-based performance profiling tool.

FLAGS

-b Suppresses the printing of the field descriptions for the Flat Profile, Call Graph Profile, and Function Index reports when they are written to a file with the **Save As** option of the File menu.

-s Produces the *gmon.sum* profile data file, if multiple *gmon.out* files are specified when Xprofiler is started. The *gmon.sum* file represents the sum of the profile information in all the specified profile files. Note that if you specify a single *gmon.out* file, the *gmon.sum* file contains the same data as the *gmon.out* file.

-z Includes functions that have both zero CPU usage and no call counts in the Flat Profile, Call Graph profile, and Function Index reports. A function will not have a call count if the file that contains its definition was not compiled with the **-pg** option, which is common with system library files.

-a Adds alternative paths to search for source code and library files, or changes the current path search order. When using this command line option, you can use the “at” symbol (@) to represent the default file path, in order to specify that other paths be searched before the default path.

-c Loads the specified configuration file. If the **-c** option is used on the command line, the configuration file name specified with it will appear in the **Configuration File (-c):** text field in the *Load Files Dialog*, and the **Selection** field of the *Load Configuration File Dialog*. When both the **-c** and **-disp_max** options are specified on the command line, the **-disp_max** option is ignored, but the value that was specified with it will appear in the **Initial Display (-disp_max):** field in the Load Files Dialog, the next time it is opened.

-disp_max
Sets the number of function boxes that Xprofiler initially displays in the function call tree. The value supplied with this flag can be any integer between 0 and 5,000. Xprofiler displays the function boxes for the most CPU-intensive functions through the number you specify. For instance, if you specify 50, Xprofiler displays the function boxes for the 50 functions in your program that consume the most CPU. After this, you can change the number of function boxes that are displayed via the Filter menu options. This flag has no effect on the content of any of the Xprofiler reports.

-e De-emphasizes the general appearance of the function box(es) for the specified function(s) in the function call tree, and limits the number of entries for these function in the Call Graph Profile report. This also applies to the specified function's descendants, as long as they have not been called by non-specified functions.

In the function call tree, the function box(es) for the specified function(s) appears greyed-out. Its size and the content of the label remain the same. This also applies to descendant functions, as long as they have not been called by non-specified functions.

In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. The information for this entry remains unchanged. Entries for descendants of the specified function do not appear unless they have been called by at least one non-specified function in the program.

-E Changes the general appearance and label information of the function box(es) for the specified function(s) in the function call tree. Also limits the number of entries for these functions in the Call Graph Profile report, and changes the CPU data associated with them. These results also apply to the specified function's descendants, as long as they have not been called by non-specified functions in the program.

In the function call tree, the function box for the specified function appears greyed-out, and its size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0 (zero). The same applies to function boxes for descendant functions, as long as they have not been called by non-specified functions. This option also causes the CPU time spent by the specified function to be deducted from the left side CPU total in the label of the function box for each of the specified function's ancestors.

In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. When this is the case, the time in the *self* and *descendants* columns for this entry is set to 0 (zero). In addition, the amount of time that was in the *descendants* column for the specified function is subtracted from the time listed under the *descendants* column for the profiled function. As a result, be aware that the value listed in the *% time* column for most profiled functions in this report will change.

-f De-emphasizes the general appearance of all function boxes in the function call tree, *except* for that of the specified function(s) and its descendant(s). In addition, the number of entries in the Call Graph Profile report for the non-specified functions and non-descendant functions is limited. The **-f** flag overrides the **-e** flag.

In the function call tree, all function boxes *except* for that of the specified function(s) and it descendant(s) appear greyed-out. The size of these boxes and the content of their labels remain the same. For the specified function(s), and it descendants, the appearance of the function boxes and labels remain the same.

In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. All information for this entry remains the same.

-F Changes the general appearance and label information of all function boxes in

the function call tree *except* for that of the specified function(s) and its descendants. In addition, the number of entries in the Call Graph Profile report for the non-specified and non-descendant functions is limited, and the CPU data associated with them is changed. The **-F** flag overrides the **-E** flag.

In the function call tree, the function box for the specified function appears greyed-out, and its size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0 (zero).

In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. The time in the *self* and *descendants* columns for this entry is set to 0 (zero). When this is the case, the time in the *self* and *descendants* columns for this entry is set to 0 (zero). As a result, be aware that the value listed in the *% time* column for most profiled functions in this report will change.

-L Uses an alternate path name for locating shared libraries. If you plan to specify multiple paths, use the *Set File Search Path* option of the File menu on the Xprofiler GUI.

-h Prints basic Xprofiler command syntax to the screen.

DESCRIPTION

Xprofiler is a GUI-based performance profiling tool, which can be used to analyze the performance of sequential as well as parallel programs. Xprofiler provides graphical function call tree display and textual profile reports to help you understand your program's CPU usage and function call counts information.

EXAMPLES

To use **xprofiler**, you first compile your program (for example, `foo.c`) with **-pg**:

```
xlc -pg -o foo foo.c
```

When the program `foo` is executed, one *gmon.out* file will be generated for each processor involved in the execution. To invoke **xprofiler**, enter:

```
xprofiler foo [[gmon.out]...]
```

FILES

/usr/lib/X11/app-defaults/Xprofiler

RELATED INFORMATION

Commands: **gprof**(1), **xlc**(1), **xlf**(1)

Appendix B. Command line flags for normal or attach mode

This appendix lists the command line flags that **poe** and **pdbx** 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 7. Command Line Flags for Normal or Attach Mode

Flag	Description	Normal Mode	Attach Mode
-procs	number of processors	yes	no
-hostfile	name of host list file	yes	no
-hfile	name of host list file	yes	no
-infolevel	message reporting level	yes	yes
-ilevel	message reporting level	yes	yes
-retry	wait for processors	yes	no
-pmlights	number of LEDs	yes	no
-usrport	port for API-to-user programmable monitor	yes	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
-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
-pgmmodel	programming model	no	no
-retrycount	retry count for node allocation	yes	no
-rmpool	default pool for job manager	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. Customizing Xprofiler 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 **xprofiler**, the IBM Parallel Environment for AIX profiling 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/Xprofiler
/usr/lib/X11/app-defaults/Xprofiler
$XAPPLRESDIR/Xprofiler
$HOME/.Xdefaults

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 Xprofiler are defined in **/usr/lib/X11/app-defaults/Xprofiler**, as well as **/usr/lpp/ppe.xprofiler/defaults/Xprofiler.ad**. In this file is a set of X resources for defining graphical user interfaces based on the following criteria:

- Window geometry
- Window title
- Push button and label text
- Color maps
- text font (in both textual reports and the graphical display).

Xprofiler resource variables

You can use the resource variables listed below to control the appearance and behavior of Xprofiler. Note that the values supplied here are the defaults, but you may change them to suit your own preferences.

Controlling fonts

To specify the font for the labels that appear with function boxes, call arcs, and cluster boxes:

Use this resource variable:	Specify this default, or a value of your own choice:
*narc*font	fixed

To specify the font used in textual reports:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*fontList	rom10

Controlling the appearance of the Xprofiler main window

To specify the size of the main window:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*mainW.height	700
Xprofiler*mainW.width	900

To specify the foreground and background colors of the main window:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*foreground	black
Xprofiler*background	light gray

To specify the number of function boxes that are displayed when you first open the Xprofiler main window:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*InitialDisplayGraph	5000

You can use the *-disp_max* command line option to override this value.

To specify the colors of the function boxes and call arcs of the function call tree:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*defaultNodeColor	forest green
Xprofiler*defaultArcColor	royal blue

To specify the color in which a specified function box or call arc is highlighted:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*HighlightNode	red
Xprofiler*HighlightArc	red

To specify the color in which de-emphasized function boxes appear:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*SuppressNode	gray

Function boxes are de-emphasized with the *-e*, *-E*, *-f*, and *-F* options.

Controlling variables related to the File menu

To specify the size of the Load Files Dialog box:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*loadFile.height	785
Xprofiler*loadFile.width	725

The Load Files Dialog box is invoked via *Load Files* option of the File menu.

To specify whether a confirmation dialog box should appear whenever a file will be overwritten:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*OverwriteOK	False

The value *True* would be equivalent to selecting the *Set Options* option from the File menu, and then selecting the *Forced File Overwriting* option from the Runtime Options Dialog box.

To specify the alternative search paths for locating source or library files:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*fileSearchPath	. (refers to current working directory)

The value you specify for *search path* is equivalent to the search path you would designate from the Alt File Search Path Dialog box. To get to this dialog box, you would choose the *Set File Search Paths* option from the File menu.

To specify the file search sequence (whether the default or alternative path is searched first):

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*fileSearchDefault	True

The value *True* is equivalent to selecting the *Set File Search Paths* from the File menu, and then the *Check default path(s) first* option from the Alt File Search Path Dialog box.

Controlling variables related to the Screen Dump option

To specify whether a screen dump will be sent to a printer or placed in a file:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*PrintToFile	True

The value *True* is equivalent to selecting the *File* button in the *Output To* field of the Screen Dump Options Dialog box. You access the Screen Dump Options Dialog box by selecting the *Screen Dump→Set Option* options from the File menu.

To specify whether the PostScript screen dump will created in grey shades or color:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*ColorPscript	False

The value *False* is equivalent to selecting the *GreyShades* button in the *PostScript Output* area of the Screen Dump Options Dialog box. You access the Screen Dump Options Dialog box by selecting the *Screen Dump→Set Option* options from the File menu.

To specify the number of grey shades that the PostScript screen dump will include (if you selected *GreyShades* in the *PostScript Output* field):

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*GreyShades	16

The value *16* is equivalent to selecting the *16* button in the *Number of Grey Shades* field of the Screen Dump Options Dialog box. You access the Screen Dump Options Dialog box by selecting the *Screen Dump→Set Option* options from the File menu.

To specify the number of seconds that Xprofiler waits before capturing a screen image:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*GrabDelay	1

The value *1* is the default for the *Delay Before Grab* option of the Screen Dump Options Dialog box, but you may specify a longer interval by entering a value here.

You access the Screen Dump Options Dialog box by selecting the *Screen Dump→Set Option* options from the File menu.

To specify the maximum number of seconds that may be specified with the slider of the *Delay Before Grab* option:

Use this resource variable:	**Specify this default, or a value of your own choice:**
Xprofiler*grabDelayScale.maximum	30

The value *30* is the default for the *Delay Before Grab* option of the Screen Dump Options Dialog box. This means that users cannot set the slider scale to a value greater than 30. You access the Screen Dump Options Dialog box by selecting the *Screen Dump→Set Option* options from the File menu.

To specify whether the screen dump is created in Landscape or Portrait format:

Use this resource variable:	**Specify this default, or a value of your own choice:**
Xprofiler*Landscape	False

The value *True* is the default for the *Enable Landscape* option of the Screen Dump Options Dialog box. You access the Screen Dump Options Dialog box by selecting the *Screen Dump→Set Option* options from the File menu.

To specify whether or not you would like information about how the image was created to be added to the PostScript screen dump:

Use this resource variable:	**Specify this default, or a value of your own choice:**
Xprofiler*Annotate	False

The value *False* is the default for the *Annotate Output* option of the Screen Dump Options Dialog box. You access the Screen Dump Options Dialog box by selecting the *Screen Dump→Set Option* options from the File menu.

To specify the directory that will store the screen dump file (if you selected *File* in the *Output To* field):

Use this resource variable:	**Specify this default, or a value of your own choice:**
Xprofiler*PrintFileName	*/tmp/Xprofiler_screenDump.ps.0*

The value you specify is equivalent to the filename you would designate in the *File Name* field of the Screen Dump Dialog box. You access the Screen Dump Options Dialog box by selecting the *Screen Dump→Set Option* options from the File menu.

To specify the printer destination of the screen dump (if you selected *Printer* in the *Output To* field):

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*PrintCommand	qprt -B ga -c -Pps

The value *qprt -B ga -c -Pps* is the default print command, but you may supply a different one here.

Controlling variables related to the View menu

To specify the size of the Overview window:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*overviewMain.height	300
Xprofiler*overviewMain.width	300

To specify the color of the highlight area of the Overview window:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*overviewGraph*defaultHighlightColor	sky blue

To specify whether the function call tree is updated as the highlight area is moved (Immediate) or only when it is stopped and the mouse button released (Delayed):

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*TrackImmed	True

The value *True* is equivalent to selecting the *Immediate Update* option from the Utility menu of the Overview window. You access the Overview window by selecting the *Overview* option from the View menu.

To specify whether the function boxes in the function call tree appear in 2-D or 3-D format:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*Shape2D	True

The value *True* is equivalent to selecting the *2-D Image* option from the View menu.

To specify whether the function call tree appears in Top-to-Bottom or Left-to-Right format:

Use this resource variable:	Specify this default, or a value of your own choice:
Xprofiler*LayoutTopDown	True

The value *True* is equivalent to selecting the *Layout: Top→Bottom* option from the View menu.

Controlling variables related to the Filter menu

To specify whether the function boxes of the function call tree are clustered or unclustered when the Xprofiler main window is first opened:

Use this resource variable:	**Specify this default, or a value of your own choice:**
Xprofiler*ClusterNode	True

The value *True* is equivalent to selecting the *Cluster Functions by Library* option from the Filter menu.

To specify whether the call arcs of the function call tree are collapsed or expanded when the Xprofiler main window is first opened:

Use this resource variable:	**Specify this default, or a value of your own choice:**
Xprofiler*ClusterArc	True

The value *True* is equivalent to selecting the *Collapse Library Arcs* option from the Filter menu.

Appendix D. Profiling programs with the AIX prof and gprof commands

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

You request parallel profiling by setting the compile flag to **-p** or **-pg** as you would with serial compilation. The parallel profiling capability of PE creates a monitor output file for each task. The files are created in the current directory, and are identified by the name *mon.out.taskid* or *gmon.out.taskid*, where *taskid* is a number between 0 and one less than the number of tasks.

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

To Profile a Serial Program:	**To Profile a Parallel Program:**
Step 1: Compile the application source code using the **cc** command with either the **-p** or **-pg** flag.	Step 1: Compile the application source code using the command **mpcc** (for C programs), **mpCC** (for C++ programs), or **mpxlf** (for Fortran programs) as described in *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment*. You should use one of the standard profiling compiler options – either **-p** or **-pg** – on the compiler command. For more information on the compiler options **-p** and **-pg**, refer to their use on the **cc** command as described in *AIX 5L Version 5.1 Commands Reference* and *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs*
Step 2: Run the executable program to produce a profile data file. If you have compiled the source code with the **-p** option, the data file produced is named *mon.out*. If you have compiled the source code with the **-pg** option, the data file produced is named *gmon.out*.	Step 2: Before you run the parallel program, set the environment variable **MP_EUILIBPATH=/usr/lpp/ppe.poe/lib/profiled:/usr/lib/profiled:/lib/profiled : /usr/lpp/ppe.poe/lib**. If your message passing library is not in **/usr/lpp/ppe.poe/lib**, substitute your message passing library path. Run the parallel program. When the program ends, it generates a profile data file for each parallel task. The system gives unique names to the data files by appending each task's identifying number to *mon.out* or *gmon.out*. If you have compiled the source code with the **-p** option, the data files produced take the form: **mon.out.***taskid* If the source code has been compiled with the **-pg** option, the data files produced take the form: **gmon.out.***taskid* **Note:** The current directory must be writable from all remote nodes. Otherwise, the profile data files will have to be manually moved to the home node for analysis with **prof** and **gprof**. You can also use the **mcpgath** command to move the files. See *IBM Parallel Environment for AIX: Operation and Use, Volume 1, Using the Parallel Operating Environment* for more about **mcpgath**.

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

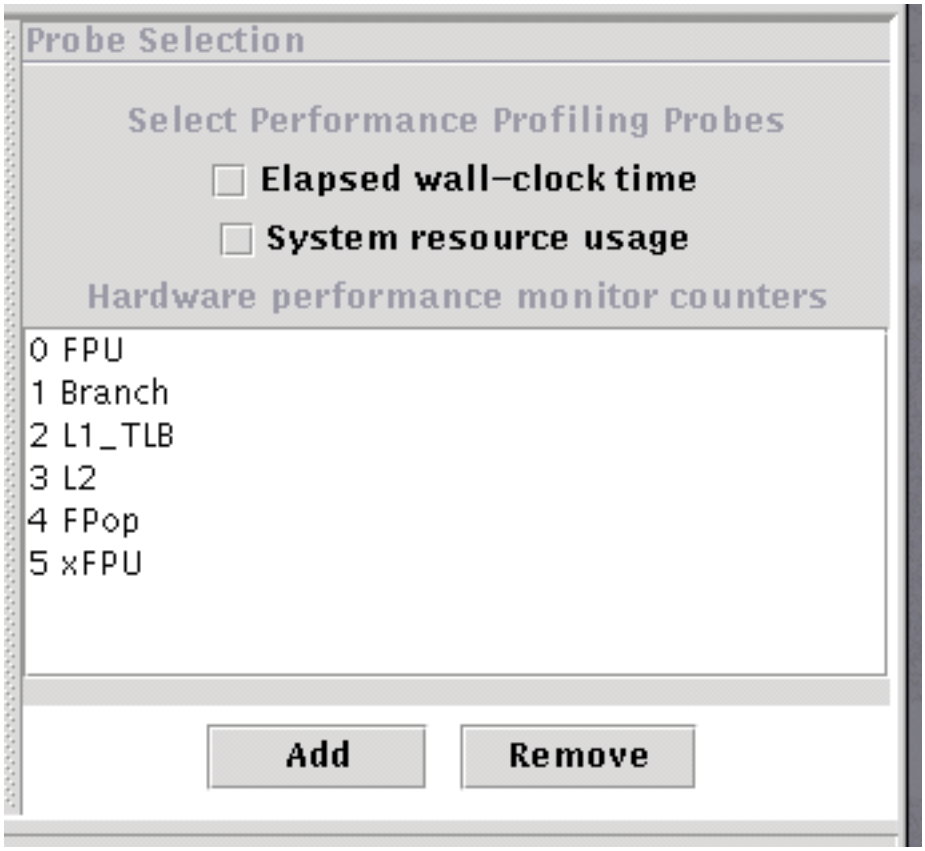
The parallel utility, **mp_profile**( ), may also be used to selectively profile portions of a program. To start profiling, call **mp_profile**(1). To suspend profiling, call **mp_profile**(0). The final profile data set will contain counts and CPU times for the program lines that are delimited by the start and stop calls. In C, the calls are **mpc_profile**(1), and **mpc_profile**(0). By default, profiling is active at the start of the user's executable.

Note: Like the sequential version of **prof**/**gprof**, if more than one profile file is specified, the parallel version of the **prof**/**gprof** command output shows the sum of the profile information in the given profile files. There is no statistical analysis contacted across the multiple profile files.

Appendix E. Understanding and Creating PCT Hardware Counter Groups

Using the Performance Collection Tool (PCT), you can add hardware counter probes to a target application to collect information on various hardware operations. We have collected similar or complementary hardware counters into groups, so that when you add one of these groups to a target application, information on different but related events are collected. The collected information can later be viewed and analyzed in the Profile Visualization Tool (PVT). The hardware counter groups available to you will differ depending on whether the processes you want to instrument are running on 604e CPUs or 630 CPUs.

If you are using the PCT's graphical user interface, the available counter groups are listed in the Select Performance Profiling Probes Panel:



If you are using the PCT's command-line interface, you can list the available counter groups by issuing the **profile show probetype hwcount** subcommand:

```
pct> profile show probetype hwcount
Prof Type Name    Description
--------- ------- ------
0         FPU     FPU, FXU, and LSU operations
1         Branch  Branch operations
2         L1_TLB  L1 cache and TLB operations
3         L2      Prefetch and L2 cache operations
4         Fpop    Floating-point operations
5         xFPU    FPU, FXU, LSU, and BPU operations
pct>
```

For more information on the Select Performance Profiling Probes Panel and the PCT's graphical user interface, refer to the PCT's online help system (as described in "Accessing the Performance Collection Tool's online help system" on page 103). For more information on the **profile show probetype hwcount** subcommand, refer to "Adding Hardware Profile Probes to Processes" on page 121.

This appendix describes:

- the specific hardware counters in each of the hardware counter groups we supply. See "Understanding the Default Hardware Counter Groups" for more information.
- how you can create your own hardware counter groups using a configuration program we supply. See "Creating Hardware Counter Groups" on page 228 for more information.

Understanding the Default Hardware Counter Groups

While you can create your own hardware counter groups, most users will be satisfied with the set of pre-defined counter groups that we supply. The following two tables describe the hardware counter groups we supply for 604e CPUs and 630 CPUs.

Table 8. Hardware counter groups for 630 CPUs

Counter group name:	**Counts:**	**Specifically, this counter group contains counters for these events:**
FPU	FPU (floating-point unit), FXU (fixed-point unit), and LSU (load/share unit) operations	• FPU0 produced a result • FPU1 produced a result • FXU0 produced a result • FXU1 produced a result • FXU2 produced a result • Number of load instructions completed • Number of store instructions completed • Processor clock cycles
Branch	Branch operations	• Branches executed • A conditional branch was predicted • Global cancel due to a branch guessed wrong • Processor clock cycles
L1_TLB	L1 cache (primary cache) and TLB (translation lookaside buffer) operations	• L1 I-cache miss • A load miss occurred in L1 • Store miss occurred in L1 • TLB miss. Includes both D-cache and I-cache misses • Snoop hit occurred and L2 has the valid block • Processor clock cycles

Table 8. Hardware counter groups for 630 CPUs (continued)

Counter group name:	Counts:	Specifically, this counter group contains counters for these events:
L2	Prefetch and L2 cache (secondary cache) operations	• Number of D-cache prefetch data stream allocations blocked due to four streams • Number of D-cache prefetch and used • RWITM caused L2 miss • Burst read caused L2 miss • Number of cycles load stalls due to interleave conflict • Processor clock cycles
FPop	Floating-point operations	• FPU0 produced a result • FPU1 produced a result • FPU divides executed (count both FPUs) • Float Multiply-Adds executed (count both FPUs) • Float Add, Multiply, Subtract executed (count both FPUs) • Number of FPU FSQRT executed (count both FPUs) • Float FCMP executed (count both FPUs) • Processor clock cycles
xFPU	FPU (floating-point unit), FXU (fixed-point unit), LSU (load/store unit), and BPU (branch processing unit) operations	• FPU0 produced a result • FPU1 produced a result • FXU0 produced a result • FXU1 produced a result • FXU2 produced a result • Branches executed • Number of load instructions completed • Number of store instructions completed

Table 9. Hardware Counter Groups for 604e CPUs

Counter group name:	Counts:	Specifically, this counter group contains counters for these events:
FPU	FPU (floating-point unit) operations	• Number of floating-point instructions completed • Processor clock cycles
FXU	FXU (fixed-point unit) operations	• Number of integer instructions completed • Processor clock cycles

Table 9. Hardware Counter Groups for 604e CPUs (continued)

Counter group name:	Counts:	Specifically, this counter group contains counters for these events:
LSU	LSU (load/share unit) operations	• Number of loads completed • LSU produced result • Processor clock cycles
Branch	Branch operations	• BPU produced result • Branch misprediction correction from execute stage • Processor clock cycles
L1	L1 cache (primary cache) operations	• Instruction cache misses • Data cache misses • Processor clock cycles
TLB	TLB (translation lookaside buffer) operations	• Number of instruction TLB misses • Data TLB misses • Processor clock cycles
Snoop	Snoop operations	• Valid snoop requests received from outside the 604e • Number of snoop hits occurred • Processor clock cycles
Loadmiss	Load miss operations	• Number of cycles a load miss takes • Processor clock cycles
Pipeline	Pipeline operations	• Number of pipeline ″flushing″ instructions • Processor clock cycles

Creating Hardware Counter Groups

In addition to our pre-defined hardware counter groups, you can create your own hardware counter groups using the *event_build* program. The *event_build* program interactively guides you through the process of selecting a hardware counter event for each counter available for a particular CPU type. The description of your counter group is saved to the file *$HOME/.pct/PMC_usergroup.conf*. The PCT will use this configuration file when listing available counter groups, and the groups you define in *PMC_usergroup.conf* will be included with the list of pre-defined hardware counter groups that we define.

To create a hardware counter group:

1. Invoke the *event_build* program. It is located in the */usr/lpp/ppe.perf/samples/user_profile* directory.

   ```
   $ /usr/lpp/ppe.perf/samples/user_profile/event_build
   ```

2. The *event_build* program prompts you to select a CPU type. Simply enter the number that corresponds with the CPU type you want to monitor. The hardware counter events and the number of counter registers available depend on the CPU type. The following example assumes that the processes you want to monitor will all be running on 604e CPUs.

```
cpu output:
[0] cpu type 604e has 4 counter
[1] cpu type 630 has 8 counter
pick a cpu or -1 to exit--> 0
```

3. The 604e CPU has four hardware counter registers available. For each of these registers, the *event_build* program asks you what event you want to monitor. Simply enter the number associated with the event you want to monitor. If you do not want to monitor anything for a particular register, enter -1. In the following example, we specify that, for the first counter register, we want to monitor the ″Number of floating-point instructions completed″. For the other three registers, we enter -1 to indicate that we don't want to monitor anything on those registers.

```
counter 1 in cpu 604e
[0] 107             PM_FPU_CMPL Number of floating-point instructions completed
[1] 203             PM_FXU_CMPL Number of integer instructions completed
[2] 304             PM_BR_MPRED Branch misprediction correction from execute stage
[3] 404             PM_LS_EXEC LSU produced result
[4] 509             PM_IC_MISS Instruction cache misses
[5] 512             PM_DTLB_MISS Data TLB misses
[6] 513             PM_SNOOP_RECV Valid snoop requests received from outside the 604e
[7] 601             PM_CYC Processor clock cycles
select a number (0 - 7) or -1 to skip  ->0
counter 2 in cpu 604e
[0] 107             PM_FPU_CMPL Number of floating-point instructions completed
[1] 203             PM_FXU_CMPL Number of integer instructions completed
[2] 304             PM_BR_MPRED Branch misprediction correction from execute stage
[3] 404             PM_LS_EXEC LSU produced result
[4] 509             PM_IC_MISS Instruction cache misses
[5] 512             PM_DTLB_MISS Data TLB misses
[6] 513             PM_SNOOP_RECV Valid snoop requests received from outside the 604e
[7] 601             PM_CYC Processor clock cycles
select a number (0 - 7) or -1 to skip  ->-1
counter 3 in cpu 604e
[0] 107             PM_FPU_CMPL Number of floating-point instructions completed
[1] 203             PM_FXU_CMPL Number of integer instructions completed
[2] 304             PM_BR_MPRED Branch misprediction correction from execute stage
[3] 404             PM_LS_EXEC LSU produced result
[4] 509             PM_IC_MISS Instruction cache misses
[5] 512             PM_DTLB_MISS Data TLB misses
[6] 513             PM_SNOOP_RECV Valid snoop requests received from outside the 604e
[7] 601             PM_CYC Processor clock cycles
select a number (0 - 7) or -1 to skip  ->-1
counter 4 in cpu 604e
[0] 107             PM_FPU_CMPL Number of floating-point instructions completed
[1] 203             PM_FXU_CMPL Number of integer instructions completed
[2] 304             PM_BR_MPRED Branch misprediction correction from execute stage
[3] 404             PM_LS_EXEC LSU produced result
[4] 509             PM_IC_MISS Instruction cache misses
[5] 512             PM_DTLB_MISS Data TLB misses
[6] 513             PM_SNOOP_RECV Valid snoop requests received from outside the 604e
[7] 601             PM_CYC Processor clock cycles
select a number (0 - 7) or -1 to skip  ->-1
generate a record:
    user: "first","my first user group","604e",107,-1,-1,-1
```

4. Once you have made your selection for each register, the *event_build* program prompts you for a name and a short description of the counter group. The name and description can be any string; and will appear in the PCT when it lists available hardware counters.

```
Enter a short name to describe it -->first
Enter a description -->my first user group
```

5. The *event_build* program prompts you to ask if you want to ″add more input (y/n)?″ If you enter ″y″, you will again be prompted for a CPU type and the

event to count on each available register. Here we enter ″n″, and the *$HOME/.pct/PMC_usergroup.conf* file is updated to include the new counter group.

```
add more input (y/n)? n
generating xml file...
    /usr/lpp/ppe.perf/samples/user_profile/xml_build < /tmp/aaaxHu0aa
```

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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
Department LJEB/P905
2455 South Road
Poughkeepsie, NY 12601-5400
U.S.A

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

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AFS

AIX

AIX/L

AIX/L (logo)

AIX 5L
e (logo)
ESCON
IBM
IBM (logo)
IBMLink
LoadLeveler
Micro Channel
pSeries
RS/6000
SP

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, MS-DOS, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

PC Direct is a trademark of Ziff Communications Company in the United States, other countries, or both and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Acknowledgments

The PE Benchmarker product includes software developed by the Apache Software Foundation, *http://www.apache.org*.

Glossary

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 application programmer that is compatible with Silicon Graphics' GL interface.

API. See *application programming interface.*

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

application programming interface. A set of programming functions and routines that provide access between the Application layer of the OSI seven-layer model and applications that want to use the network. It is a software interface.

argument. A parameter passed between a calling program and a called program or subprogram.

attribute. A named property of an entity.

B

bandwidth. The difference, expressed in hertz, between the highest and the lowest frequencies of a range of frequencies. For example, analog transmission by recognizable voice telephone requires a bandwidth of about 3000 hertz (3 kHz). The bandwidth of an optical link designates the information-carrying capacity of the link and is related to the maximum bit rate that a fiber link can support.

blocking operation. An operation that does not complete until the operation either succeeds or fails. For example, a blocking receive will not return until a message is received or until the channel is closed and no further messages can be received.

breakpoint. A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

broadcast operation. A communication operation 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 Uniforum in 1983 and the ANSI standards committee for the C language in 1984.

C++. A general-purpose programming language that is based on the C language. C++ includes extensions that support an object-oriented programming paradigm. Extensions include:

- strong typing
- data abstraction and encapsulation
- polymorphism through function overloading and templates
- class inheritance.

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 that 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. A cluster typically provides excellent price/performance.

collective communication. A communication operation that involves more than two processes or tasks. Broadcasts, reductions, and the **MPI_Allreduce** subroutine are all examples of collective communication operations. All tasks in a communicator must participate.

command alias. When using the PE command line debugger **pdbx**, you can create abbreviations for existing commands using the **pdbx alias** command. These abbreviations are known as *command aliases.*

Communication Subsystem (CSS). A component of the IBM Parallel System Support Programs for AIX that provides software support for the SP Switch. CSS provides two protocols: Internet Protocol (IP) for LAN-based communication and user space (US) 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 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 a severe user error. The current program state is needed for the programmer to diagnose and correct the problem.

core file. A file that preserves the state of a program, usually just before a program is terminated for an unexpected error. See also *core dump.*

current context. When using the **pdbx** debugger, control of the parallel program and the display of its data can be limited to a subset of the tasks belonging to that program. This subset of tasks is called the *current context.* You can set the current context to be a single task, multiple tasks, or all the tasks in the program.

D

data decomposition. A method of breaking up (or decomposing) a program into smaller parts to exploit parallelism. One divides the program by dividing the data (usually arrays) into smaller parts and operating on each part independently.

data parallelism. Refers to situations where parallel tasks perform the same computation on different sets of data.

dbx. A symbolic command line debugger that is often provided with UNIX systems. The PE command line debugger **pdbx** is based on the **dbx** debugger.

debugger. A debugger provides an environment in which you can manually control the execution of a program. It also provides the ability to display the program' data and operation.

distributed shell (dsh). A PSSP command that lets you issue commands to a group of hosts in parallel. See *IBM Parallel System Support Programs for AIX: Command and Technical Reference* for details.

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

Ethernet. A baseband local area network (LAN) that allows multiple stations to access the transmission medium at will without prior coordination, avoids contention by using carrier sense and deference, and resolves contention by using collision detection and delayed retransmission. Ethernet uses carrier sense multiple access with collision detection (CSMA/CD).

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

executable. A program that has been link-edited and therefore can be run in a processor.

execution. To perform the actions specified by a program or a portion of a program.

expression. In programming languages, a language construct for computing a value from one or more operands.

F

fairness. A policy in which tasks, threads, or processes must be allowed eventual access to a resource for which they are competing. For example, if multiple threads are simultaneously seeking a lock, no set of circumstances can cause any thread to wait indefinitely for access to the lock.

FDDI. See *Fiber Distributed Data Interface.*

Fiber Distributed Data Interface (FDDI). An American National Standards Institute (ANSI) standard for a local area network (LAN) using optical fiber cables. An FDDI LAN can be up to 100 kilometers (62 miles) and can include up to 500 system units. There can be up to 2 kilometers (1.24 miles) between system units and concentrators.

file system. 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 functions. Updates correct an error in, or enhance, a previously installed program. 2) One or more separately-installable, logically-grouped units in an installation package. See also *licensed program* and *package.*

foreign host. See *remote host.*

FORTRAN. One of the oldest of the modern programming languages, and the most popular language for scientific and engineering computations. Its name is a contraction of *FORmula TRANslation.* The two most common FORTRAN versions are FORTRAN 77, originally standardized in 1978, and FORTRAN 90. FORTRAN 77 is a proper subset of FORTRAN 90.

function 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 method is in contrast to data decomposition, which distributes the same work over different data to independent processors.

functional parallelism. Refers to situations where parallel tasks specialize in particular work.

G

Gauss-Seidel. An iterative relaxation method for solving Laplace's equation. It calculates the general solution by finding particular solutions to a set of discrete points distributed throughout the area in question. The values of the individual points are obtained by averaging the values of nearby points. Gauss-Seidel differs from Jacobi-Seidel in that, for the i+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, COBOL, FORTRAN, or Pascal programs. The execution profile is in a textual and tabular format. It is useful for identifying which routines use the most CPU time. See the man page on **gprof**.

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

GUI. See *Graphical user interface.*

H

HIPPI. High performance parallel interface.

hook. A **pdbx** command that lets you re-establish control over all tasks in the current context that were previously unhooked with this command.

home node. The node from which an application developer compiles and runs his program. The home node can be any workstation on the LAN.

host. A computer connected to a network that provides an access method to that network. A host provides end-user services.

host list file. A file that contains a list of host names, and possibly other information, that was defined by the application 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 (PE). A licensed program that provides an execution and development

environment for parallel C, C++, and FORTRAN programs. PE 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*, and *package.*

IBM Parallel System Support Programs for AIX (PSSP). A comprehensive suite of applications that is used to manage an RS/6000 SP system as a full-function parallel-processing system. PSSP provides a single point of control for administrative tasks and helps increase productivity by letting administrators view, monitor, and control how the system operates.

Internet. The collection of worldwide networks and gateways that 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 SP Switch library, provided with the IBM Parallel System Support Programs for AIX, that follows the IP protocol of TCP/IP.

IP. Internet Protocol.

J

Jacobi-Seidel. See *Gauss-Seidel.*

Jumpshot. A public domain tool, developed at Argonne National Laboratory, for visualizing program performance. Jumpshot reads files in a scalable log file format, called SLOG. The PE Benchmarker toolset provides the **slogmerge** utility to enable you to convert UTE files into the SLOG format.

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 that controls the resources of the CPU and allocates them to the users. The kernel is memory-resident, is said to run in *kernel mode* (in other words, at higher execution priority level than *user mode*), and is protected from user tampering by the hardware.

L

LAPI. See *low-level communication API.*

Laplace's equation. A homogeneous partial differential equation used to describe heat transfer, electric fields, and many other applications.

The dimension-free version of Laplace's equation is:

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. A collection of software packages sold as a product that customers pay for to license. A licensed program 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.*

lightweight corefiles. An alternative to standard AIX corefiles. Corefiles produced in the *Standardized Lightweight Corefile Format* provide simple process stack traces (listings of function calls that led to the error) and consume fewer system resources than traditional corefiles.

LoadLeveler. A job management system that works with POE to let users run jobs and match processing needs with system resources, in order to make better use of the system.

local variable. A variable that is defined and used only in one specified portion of a computer program.

loop unrolling. A program transformation that makes multiple copies of the body of a loop, also placing the copies within the body of the loop. The loop trip count and index are adjusted appropriately so the new loop computes the same values as the original. This transformation makes it possible for a compiler to take additional advantage of instruction pipelining, data cache effects, and software pipelining.

See also *optimization.*

low-level communication API (LAPI). A low-level (low overhead) message-passing protocol that uses a one-sided, active-message-style interface to transfer messages between processes. LAPI is an IBM proprietary interface that exploits the SP Switch adapters.

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.

message passing client interface (MPCI). The primary interface to the point¡to¡point message-passing protocols that support the SP Switch and the SP Switch2.

message passing interface (MPI). An industry-standard message-passing protocol that typically uses a two-sided send-receive model to transfer messages between processes.

MIMD. See *multiple instruction stream, multiple data stream.*

MPCI. See *message passing client interface.*

MPMD. See *Multiple program, multiple data.*

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

MPI. See *message passing interface.*

multiple instruction stream, multiple data stream (MIMD). A parallel programming model in which different processors perform different instructions on different sets of data.

N

netCDF file. See *network Common Data Form (netCDF) file*.

network. An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

network Common Data Form (netCDF) file. A file using the netCDF format developed at the Unidata Program Center — a program funded by the National Science Foundation (NSF). In the PE Benchmarker toolset, the PVT reads netCDF files containing hardware and operating system profiles output by the PCT.

Network Information Services (NIS). A set of UNIX network services (for example, a distributed service for retrieving information about the users, groups, network addresses, and gateways in a network) that resolve naming and addressing differences among computers in a network.

NIS. See *Network Information Services.*

node. (1) In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network. (2) 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, that 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. Contrast with *source code.*

optimization. A widely-used (though not strictly accurate) term for program performance improvement, especially for performance improvement done by a compiler or other program translation software. An optimizing compiler is one that performs extensive code transformations in order to obtain an executable that runs faster but gives the same answer as the original. Such code transformations, however, can make code debugging and performance analysis very difficult because complex code transformations obscure the correspondence between compiled and original source code.

option flag. Arguments or any other additional information that a user specifies with a program name. Also referred to as *parameters* or *command line options.*

P

package. A number of filesets that have been collected into a single installable image of licensed programs. Multiple filesets can be bundled together for installing groups of software together. See also *fileset* and *licensed program.*

Parallel Environment. See *IBM Parallel Environment for AIX.*

parallelism. The degree to which parts of a program may be concurrently executed.

parallelize. To convert a serial program for parallel execution.

Parallel System Support Programs. See *IBM Parallel System Support Programs for AIX.*

Parallel Operating Environment (POE). An execution environment that smooths the differences between serial and parallel execution. POE lets you submit and manage parallel jobs.

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).

PCT. See *Performance Collection Tool.*

pdbx. The parallel, symbolic command line debugging facility of PE. **pdbx** is based on the **dbx** debugger and has a similar interface.

PE. See *IBM Parallel Environment for AIX.*

PE Benchmarker. A suite of applications and utilities that you can use to analyze the performance of programs run within IBM Parallel Environment for AIX. The PE Benchmarker toolset consists of the Performance Collection Tool (PCT), the Profile Visualization Tool (PVT), and a set of Unified Trace Environment (UTE) utilities.

Performance Collection Tool (PCT). Part of the PE Benchmarker toolset, this tool enables you to collect either MPI and user event data or hardware and operating system profiles for one or more application processes. Because it is built on dynamic instrumentation technology, the PCT enables you to insert instrumentation probes into a target application while the target application is running.

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

PID. See *process identifier.*

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.

POE. See *Parallel Operating Environment.*

pool. Groups of nodes on an SP that are known to LoadLeveler, and are identified by a pool name or number.

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.

process identifier (PID). An integer used by the Unix kernel to uniquely identify a process. PIDs are returned by the **fork system** call and can be passed to **wait()** or **kill()** to perform actions on the given process.

prof. A utility that produces an execution profile of an application or program. **prof** is useful for identifying which routines use the most CPU time.

Profile Visualization Tool (PVT). Part of the PE Benchmarker toolset, this tool reads the hardware or operating system profiles output by the PCT in netCDF format. The PVT enables you to summarize the collected information in reports.

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.

PSSP. See *IBM Parallel System Support Programs for AIX.*

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

PVT. See *Profile Visualization Tool.*

R

reduced instruction set computer (RISC). A computer that uses a small, simplified set of frequently-used instructions for rapid execution.

reduction operation. An operation, usually mathematical, that reduces a collection of data by one or more dimensions. For example, the arithmetic SUM operation is a reduction operation 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. Xprofiler generates five types of reports. Each type of report presents different statistical information for an application.

RISC. See *reduced instruction set computer.*

RS/6000 SP. A scalable parallel system arranged in various physical configurations that provides a high-powered computing environment.

S

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

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

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

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.

single program, multiple data (SPMD). A parallel programming model in which different processors execute the same program on different sets of data.

SLOG file. A file using the scalable log file format. The Jumpshot tool from Argonne National Laboratory reads files of this format. The **slogmerge** utility of the PE Benchmarker toolset converts UTE files into the SLOG file format.

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

source line. A line of source code.

SP. See *RS/6000 SP.*

SPMD. See *single program, multiple data.*

SP Switch. The high-performance message-passing network of the RS/6000 SP system that connects all processor nodes.

standard error (STDERR). (1) An output file intended to be used for error messages for C programs. (2) In many workstation-based operating systems, the output stream to which error messages or diagnostic messages are sent.

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.

STDERR. See *standard error.*

STDIN. See *standard input.*

STDOUT. See *standard output.*

stencil. A pattern of memory references used for averaging. A 4-point stencil in two dimensions for a given array cell, x(i,j), uses the four adjacent cells, x(i-1,j), x(i+1,j), x(i,j-1), and x(i,j+1).

subroutine. (1) A sequence of instructions whose execution is invoked by a call. (2) A sequenced set of instructions or statements that 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.

T

target application. See *DPCL target application.*

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 information about the execution of the program. This information is accumulated into a trace file that can later be examined.

tracepoint. Tracepoints are places in the program that, when reached during execution, cause the debugger to print information about the state of the program.

trace record. In PE, a collection of information about a specific event that occurred during the execution of your program. For example, a trace record is created for each send and receive operation that occurs in your program (this is optional and may not be appropriate). These records are then accumulated into a trace file that can later be examined.

U

Unified Trace Environment (UTE). An environment that is used to generate, analyze, and visualize trace events for applications running on IBM RS/6000 SP systems.

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 Switch, that maximizes the performance capabilities of the SP hardware.

UTE. See *Unified Trace Environment.*

UTE interval files. A Unified Trace Environment file that is distinct from an AIX trace file by the inclusion of interval information. While an AIX trace file has a time stamp indicating the point in time when an event occurred, UTE interval files also determine how long an event lasts before encountering the next event. Because they include this duration information, UTE interval files are easier to visualize than traditional AIX event trace files.

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) To display and look at data on screen.

(2) A special display of data, created as needed. A view temporarily ties two or more files together so that the combined files can be displayed, printed, or queried. The user specifies the fields to be included. The original files are not permanently linked or altered; however, if the system allows editing, the data in the original files will be changed.

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.

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.

Bibliography

This bibliography helps you find product documentation related to the RS/6000 SP hardware and software products.

You can find most of the IBM product information for RS/6000 SP products on the World Wide Web. Formats for both viewing and downloading are available.

PE documentation is shipped with the PE licensed program in a variety of formats and can be installed on your system. See “Accessing PE documentation online” and “Parallel Environment (PE) publications” on page 244 for more information.

This bibliography also contains a list of non-IBM publications that discuss parallel computing and other topics related to the RS/6000 SP.

Information formats

Documentation supporting RS/6000 SP software licensed programs is no longer available from IBM in hardcopy format. However, you can view, search, and print documentation in the following ways:

- On the World Wide Web
- Online (on the product media and via the SP Resource Center)

Finding documentation on the World Wide Web

Most of the RS/6000 SP hardware and software books are available from the IBM Web site at:

http://www.ibm.com/servers/eserver/pseries

The serial and parallel programs that you find in the *IBM Parallel Environment for AIX: Hitchhiker's Guide* are also available from this Web site, in the same location as the PE online library.

You can view a book, download a Portable Document Format (PDF) version of it, or download the sample programs from the *IBM Parallel Environment for AIX: Hitchhiker's Guide*.

At the time this manual was published, the Web address of the *RS/6000 SP Product Documentation Library* page was:

http://www.rs6000.ibm.com/resource/aix_resource/sp_books

However, the structure of the IBM Web site may change over time.

Accessing PE documentation online

On the same medium as the PE product code, IBM ships PE man pages, HTML files, and PDF files. To use the PE online documentation, you must first install these filesets:

- **ppe.html**
- **ppe.man**
- **ppe.pdf**

To view the PE HTML publications, you need access to an HTML document browser such as Netscape. The HTML files and an index that links to them are installed in the **/usr/lpp/ppe.html** directory. Once the HTML files are installed, you can also view them from the RS/6000 SP Resource Center.

To view the PE PDF publications, you need access to the Adobe Acrobat Reader. The Acrobat Reader is shipped with the AIX Bonus Pack and is also freely available for downloading from the Adobe Web site at:

http://www.adobe.com

To successfully print a large PDF file (approximately 300 or more pages) from the Adobe Acrobat reader, you may need to select the "Download Fonts Once" button on the Print window.

If you have installed the SP Resource Center on your SP system, you can access it by entering this command:

```
/usr/lpp/ssp/bin/resource_center
```

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

RS/6000 SP publications

SP planning publications

The following publications are related to this book only if you run parallel programs on the RS/6000 SP. These books are not related if you use a network cluster that is made up of IBM @server pSeries processors, IBM RS/6000 processors, or both.

- *IBM RS/6000 SP: Planning, Volume 1, Hardware and Physical Environment*, GA22-7280
- *IBM RS/6000 SP: Planning, Volume 2, Control Workstation and Software Environment*, GA22-7281

SP software publications

GPFS publications

- *IBM General Parallel File System for AIX: Administration and Programming Reference*, SA22-7452
- *IBM General Parallel File System for AIX: Concepts, Planning, and Installation Guide*, GA22-7453
- *IBM General Parallel File System for AIX: Data Management API Guide*, GA22-7435
- *IBM General Parallel File System for AIX: Problem Determination Guide*, GA22-7434

LoadLeveler publications

- *IBM LoadLeveler for AIX 5L: Diagnosis and Messages Guide*, GA22-7277
- *IBM LoadLeveler for AIX 5L: Using and Administering*, SA22-7311

Parallel Environment (PE) publications

- *IBM Parallel Environment for AIX: Hitchhiker's Guide*, SA22-7424
- *IBM Parallel Environment for AIX: Installation*, GA22-7418

- *IBM Parallel Environment for AIX: Messages*, GA22-7419
- *IBM Parallel Environment for AIX: MPI Programming Guide*, SA22-7422
- *IBM Parallel Environment for AIX: MPI Subroutine Reference*, SA22-7423
- *IBM Parallel Environment for AIX: Operation and Use, Volume 1*, SA22-7425
- *IBM Parallel Environment for AIX: Operation and Use, Volume 2*, SA22-7426

PSSP publications

The following publications are related to this book only if you run parallel programs on the RS/6000 SP. These books are not related if you use a network cluster that is made up of IBM @server pSeries processors, IBM RS/6000 processors, or both.

- *IBM Parallel System Support Programs for AIX: Administration Guide*, SA22-7348
- *IBM Parallel System Support Programs for AIX: Command and Technical Reference*, SA22-7351
- *IBM Parallel System Support Programs for AIX: Diagnosis Guide*, GA22-7350
- *IBM Parallel System Support Programs for AIX: Installation and Migration Guide*, GA22-7347
- *IBM Parallel System Support Programs for AIX: Messages Reference*, GA22-7352
- *IBM Parallel System Support Programs for AIX: Planning, Volume 2*, GA22-7281
- *IBM Parallel System Support Programs for AIX: Managing Shared Disks*, SA22-7349

RS/6000 Cluster Technology (RSCT) publications

- *IBM RS/6000 Cluster Technology: Event Management Programming Guide and Reference*, SA22-7354
- *IBM RS/6000 Cluster Technology: Group Services Programming Guide and Reference*, SA22-7355
- *IBM RS/6000 Cluster Technology: First Failure Data Capture Programming Guide and Reference*, SA22-7454

AIX publications

You can find links to the latest AIX publications on the IBM Web site at:

http://www.ibm.com/servers/aix/library/techpubs.html

DCE publications

You can view a DCE book or download a PDF version of it from the IBM Web site at:

http://www.ibm.com/software/network/dce/library

Red books

IBM's International Technical Support Organization (ITSO) has published a number of redbooks related to the RS/6000 SP. For a current list, see the IBM Web site at:

http://www.ibm.com/redbooks

Non-IBM publications

Here are some non-IBM publications that you might find helpful.

- Almasi, G. and A. Gottlieb. *Highly Parallel Computing*, Benjamin-Cummings Publishing Company, Inc., 1989.
- Bergmark, D., and M. Pottle. *Optimization and Parallelization of a Commodity Trade Model for the SP1*. Cornell Theory Center, Cornell University, June 1994.
- Foster, I. *Designing and Building Parallel Programs*, Addison-Wesley, 1995.
- Gropp, William, Ewing Lusk, and Anthony Skjellum. *Using MPI*, The MIT Press, 1994.

  As an alternative, you can use SR28-5757 to order this book through your IBM representative or IBM branch office serving your locality.
- Koelbel, Charles H., David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance FORTRAN Handbook*, The MIT Press, 1993.
- Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 1.1*, University of Tennessee, Knoxville, Tennessee, June 6, 1995.
- Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface, Version 2.0*, University of Tennessee, Knoxville, Tennessee, July 18, 1997.
- Pfister, Gregory, F. *In Search of Clusters*, Prentice Hall, 1998.
- Snir, M., Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference* The MIT Press, 1996.
- Spiegel, Murray R. *Vector Analysis* McGraw-Hill, 1959.

For more information about the Message Passing Interface Forum and the MPI standards documents, see:

http://www.mpi-forum.org

Index

Q

R

S

Readers' comments – We'd like to hear from you

IBM Parallel Environment for AIX
Operation and Use, Volume 2
Version 3 Release 2

Publication No. SA22-7426-01

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name _____ Address _____

Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You
SA22-7426-01

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie NY 12601-5400

Fold and Tape | **Please do not staple** | Fold and Tape

SA22-7426-01

Cut or Fold
Along Line

Program Number: 5765-D93

SA22-7426-01